A Thesis for the Degree of Ph.D. in Engineering

Making GPUs First-Class Citizen Computing Resources in Multi-Tenant Cloud Environments

August 2018

Graduate School of Science and Technology

Keio University

Yusuke Suzuki

Acknowledgement

I would like to thank my advisor, Prof. Kenji Kono. His constant guidance helped me in all the time of research. I would like to express my sincere gratitude to Prof. Hiroshi Yamada and Prof. Shinpei Kato for their invaluable insight. This dissertation would not have been possible without their advice and encouragement.

I am grateful to the members of my thesis committee as well: Prof. Nobuyuki Yamasaki, Prof. Hiroki Matsutani, and Prof. Kenichi Kourai. This dissertation was greatly improved by their valuable feedback.

I am also thankful to my colleagues in the sslab. Their surprising enthusiasm and skills have always inspired me.

I appreciate the financial supports from the Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists and the Core Research for Evolutional Science and Technology of Japan Science and Technology Agency.

Finally, I thank my family, my parents, sister, and brother for their support all these years. Without their support and encouragement, many accomplishments in my life including this dissertation would not have been possible.

Abstract

Making GPUs First-Class Citizen Computing Resources in Multi-Tenant Cloud Environments

Graphic processing units (GPUs) provide massively parallel computational power and encourage the use of general-purpose computing on GPUs (GPGPU). GPGPU has become an attractive platform in various domains of applications including server-side workloads. Adaption of GPGPU in server-side workloads and scaling up of GPU computing capacity motivate the consolidation of GPGPU applications. Making GPUs first-class citizen computing resources in the cloud is a key to consolidation in multi-tenant cloud platforms. Despite the previous study on GPU resource virtualization, the tradeoffs between the approaches remain unclear. Shedding light on these tradeoffs and the technical requirements for the resource virtualization at various interface-levels would facilitate the development of an appropriate GPU resource virtualization solution.

This dissertation presents two approaches for GPU resource virtualization, GPUvm and GLoop. GPUvm is an architecture for hypervisor-level GPU virtualization. GPUvm offers three modes: the full-, naive para-, and high-performance para-virtualization. GPUvm exposes low- and high-level interfaces such as memory-mapped I/O and DRM APIs to the guest virtual machines (VMs). Our experiments show that GPUvm incurs different overheads as the level of the exposed interfaces is changed. The results also show that GPU scheduling can achieve a coarse-grained fairness among multiple VMs.

We also present GLoop, a software runtime that enables us to consolidate GPGPU applications including advanced GPU applications. While the coarse-grained fairness can be achieved by the application-transparent approaches, advanced GPGPU applications, referred to as GPU eaters, can monopolize a shared

GPU. GLoop explores the way to achieve consolidation of GPU eaters by taking an application-assisted approach including modification of the applications. GLoop introduces an event-driven programming model to offer the GPU eaters' high functionality while scheduling them on a shared GPU with a proportional-share policy. We implement a prototype of GLoop and port eight GPU eaters on it. Our experiments show that our prototype successfully schedules the consolidated GPGPU applications on the basis of its scheduling policy and isolates resources among them.

The contribution of this dissertation is twofold. First, we show the design and implementation of full-virtualized GPUs, clarify the bottleneck, and show that the high-level interface for virtual GPUs can mitigate the overheads. This helps the cloud software developers to select an appropriate virtualization approach for their use cases, and helps GPU hardware vendors to design the future GPU hardware extension for virtualization. Second, we show the limitation of the application-transparent approaches, and show that the application-assisted approach can share a GPU even in the face of GPU eaters. This allows the multi-tenant clouds to share a GPUs with a wider range of applications. Moreover, GLoop envisions the clouds sharing not only GPUs but also other non-preemptive accelerators.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphic Processing Units (GPUs) become distinguished accelerators for a broad range of applications because of their significant performance benefit and high energy efficiency. GPUs are composed of thousands of compute cores, which characterize GPUs as accelerators for massively data-parallel computations. Beyond the graphic purpose, general-purpose computing on GPUs (GPGPU) becomes widely accepted technique in various application domains, which include deep learning [1, 19, 39, 59], scientific simulations [52, 73], file systems [74, 78], complex control systems [41, 66], autonomous vehicles [37, 53], and server applications including network systems [33, 38], web servers [3], key-value stores [35] and databases [14, 34, 40, 46, 51, 69, 87].

Making GPUs a first-class citizen computing resource is a critical requirement for hosting GPGPU applications in multi-tenant cloud platforms whose resources are shared among multiple users. Increasing adoption of GPGPU in server workloads leads to making GPUs stock keeping units in cloud platforms. GPGPU in server workloads motivates consolidating GPGPU applications on shared GPUs in the cloud. For example, since the load of cloud services varies with diurnal patterns and spikes [13], GPGPU server consolidation can improve GPU utilization by assigning the idle-time of the GPU to not only other GPGPU servers but also compute-intensive GPGPU applications including those of deep learning. The motivation for consolidation is strengthened by the fact that GPUs are continuously scaling up. NVIDIA has reported that the number of streaming processors and size of memory in Tesla M40 GPUs are 1.6 times and 2.0 times larger than those of the previous generation [61].

Resource and performance isolation are keys for sharing GPUs in multi-tenant cloud platforms. Without resource isolation, multiple tenants cannot share a GPU in a secure manner. Since GPUs are recognized as I/O devices from the rest of the systems, GPU computing resources cannot be shared among multiple tenants without first virtualizing them as computing accelerators. Supporting virtualization of GPU computing resources enables a GPU to be isolated among the virtual machines (VMs) or containers, which are used as the logical unit of the computing resources in a cloud.

Performance isolation, in particular, time-multiplexing of GPU computing resources poses a new challenge in multi-tenant consolidation. GPGPU applications use GPU computing resources by launching GPU kernels that are routines executed on GPUs. Recent high-functioning GPGPU applications, referred to as GPU eaters, typically launch a long- or infinite-running GPU kernel and monopolize a shared GPU, easily starving other GPGPU applications collocated on it. For example, GPUfs- [74] and GPUnet-based [48] applications poll completions of I/O requests on the GPU. Scientific applications [52, 73] exclusively use GPUs to compute their simulations.

Throughout this dissertation, we focus on *discrete* GPUs. *Discrete* GPUs are widely accepted for their intensive computational ability when compared with *integrated* GPUs in the GPGPU field. GPUs are classified into *discrete* (on-board and off-chip) and *integrated* (on-chip) GPUs. Discrete GPUs are connected on the PCI express bus (PCIe) and are composed of a huge number of cores tightly coupled with a specialized high bandwidth device memory, while integrated GPUs reside on the same chip as the CPUs and share system memory with CPUs. As discussed in [48], the discrete GPU design delivers a greater computational performance and a higher energy efficiency, whereas integrated GPUs are oriented to a lower latency and a lower thermal design power (TDP). The more recent research has leveraged discrete GPUs to create high-performance, scalable, and more energy efficient cloud applications [48, 68, 77].

1.1 Previous Approaches

We summarize the pros and cons of the current approaches to sharing GPUs. Resource virtualization of GPU is categorized into four types: multiplexing in Operating Systems (OSes), hypervisors, hardwares, and application-assisted approaches.

Approaches at the OSes and hypervisors virtualize GPU devices or their computing capabilities, and do not require application modifications. Application-assisted approaches require developers to modify applications or use specific frameworks to make GPU sharing available. While application-assisted approaches require support from applications, fine-grained control (e.g. time-multiplexing of GPU kernels) can be achieved.

1.1.1 Multiplexing GPUs at OS

Multiplexing GPUs at the OSes is useful in multi-programming environments where a user concurrently runs multiple GPU applications including games, video players, and so on. By modifying GPU device drivers [43], introducing kernel modules [44, 55], or introducing different system calls and programming paradigm [67], these approaches achieve coarse-grained performance isolation among multiple GPU applications. The OSes and GPU drivers isolate applications by using the process abstraction. Since these approaches are done at the OSes, simply bringing these approaches to the cloud using hypervisors is not possible.

1.1.2 GPU Virtualization at Hypervisor

The approaches of GPU resource virtualization at the hypervisor are classified into I/O pass-through, API remoting, hybrid, or mediated pass-through. These approaches are also referred to as *back-end*, *front-end*, *para*, and *full* virtualization, respectively [22].

I/O pass-through [6] directly exposes the GPU hardware to guest device drivers. The virtualization extension for directed I/O such as Intel VT-d [2] allows hypervisors to assign devices to guest VMs without compromising isolation.

This can provide close to a native performance, but a physical GPU is assigned to a single VM by hardware design.

API remoting [5, 23, 26, 31, 32, 47, 50, 72, 85, 86] is more suitable for multi-tasking and is relatively easy to implement. A high-level API such as CUDA in this approach is exported to the guest VMs by installing a wrapper library. The API calls from the guest VMs through a wrapper library are routed to the server owning the GPUs, and then, the server invokes the APIs through the original library. Although this approach is simple, it lacks flexibility in the choice of languages and libraries and can cause a version incompatibility between a wrapper library and an original library. The entire software stack must be rewritten to incorporate an API remoting mechanism. Implementing API remoting could also result in enlarging the trusted computing base (TCB) due to the need to accommodate for additional libraries and drivers in the server.

Para-virtualization [22] provides an ideal device model through the hypervisor and allows multiple VMs to concurrently access the GPU. It can provide lower-level control to the guest drivers than in API remoting and minimizes the overhead of the virtualization, but the guest device drivers must be modified to support the device model.

Full-virtualization [82] enables for multiplexing without needing any drivers or runtime modification. It allows guest VMs to use vanilla device drivers while providing resource isolation on multiple VMs for GPGPU. These features are attractive to IaaS environments on which the users can use existing GPGPU software stacks without any guest modifications. However, to the best of our knowledge, no designs or evaluations of the full-virtualization for discrete GPUs have been reported. gVirt [82] enables for the full-virtualization of the Intel integrated GPUs, but they have different hardware designs than those for discrete GPUs. gVirt also changes the specifications and driver of Intel Integrated GPUs.

1.1.3 Application-assisted Approaches

While the approaches in Section 1.1.1 and Section 1.1.2 typically do not require modifications of applications, they fail to schedule GPU eater's GPU kernels if GPUs are non-preemptive. GPU kernel launchers [44, 67] schedule GPU kernels from GPGPU applications. GPU command-based schedulers [43,55,79] schedule

GPGPU applications at the boundary of GPU commands instead of GPU kernels. GPU device drivers submit GPU commands to drive GPUs, so GPU commands are low-level interface to GPU devices. However, these schedulers fail to schedule GPU eaters, which have long- or infinite-running GPU kernels since launching a GPU kernel is represented as one command.

A naive application-assisted approach to scheduling GPU eaters in an isolated manner is to divide the GPU eater's kernels into smaller GPU kernels by splitting the GPU computations and finishing all the running thread blocks. This approach, called kernel splitting, offers scheduling points to typical GPU schedulers that use GPU kernel launches as scheduling points. However, it degrades the performance of the GPU kernels due to the high cost of kernel launches.

Thread block schedulers [16, 64, 84] schedules thread blocks that compose a GPU kernel. These approaches use the ends of thread blocks as scheduling points. Thus, even with them, a GPU eater with long-running thread blocks can still monopolize a shared GPU.

Other techniques, such as context funneling [60, 83] and persistent threads [30], effectively schedule GPU eaters but fail to isolate GPGPU applications since they run GPU kernels in a shared GPU context where all the kernels share the same address space; thus, a hosted GPGPU application may access and modify the memory of other GPGPU applications, which is not suitable for multi-tenant cloud platforms.

1.1.4 Hardware Preemption

Current hardware preemption is not a perfect solution to consolidate GPU eaters in multi-tenant cloud platforms. The recent NVIDIA Pascal GPUs [61] have mechanisms to preempt long-running GPU kernels. However, as recent literature [84] reported, no publicly available information shows the availability of software-level preemption control. Because of the lack of software control, we cannot apply a proportional share policy to GPU kernels that is based on various indicators such as customer payment. Therefore, if a user starts many GPU contexts, this user can simply occupy the GPU's computing resources. In addition, if a GPU eater polls I/O completion, GPU cycles are wasted because the hardware-level scheduler assigns timeslices to it without recognizing the polling.

1.2 Motivation

Despite all the study on the resource virtualization of GPUs, the important tradeoffs between the approaches remain unclear because of a lack of designs for and quantitative evaluations of resource virtualization approaches. First, the design, overhead, and bottleneck of full-virtualization are not explored, which prevent the cloud vendors from selecting appropriate virtualization approaches for their cloud. Second, the application-assisted approach for GPU eaters is not shown, which limits applicability of GPU sharing in the multi-tenant cloud environments. The approaches have different tradeoffs in terms of performance, functionalities, requirement of application modification, and limitations. Exploring these approaches clarifies the tradeoffs and technical difficulties and allows cloud software developers and hardware designers to design and discuss efficient virtualization solutions.

1.3 Study Overview

In this dissertation, we explore the tradeoffs between the approaches of GPU resource virtualization. We tackle the resource isolation and time-multiplexing of GPU kernel execution. To achieve this goal, we explore two approaches, hypervisor-level GPU virtualization and application-assisted approaches.

We show GPUvm, a hypervisor-level GPU virtualization approach. GPUvm offers three types of hypervisor-level GPU virtualization: full-, naive para-, and high-performance para-virtualization. In the full- and naive para-virtualizations, we expose a native GPU device model to provide a low-level interface through memory-mapped I/O (MMIO). In the naive para-virtualization, we provide a hypercall interface to mitigate the major source of overhead in the full-virtualization. In high-performance para-virtualization, which is called PVDRM, we expose the high-level interface. PVDRM leverages the *Direct Rendering Manager* (DRM) APIs as an interface. DRM is a widely used GPU abstraction layer in Linux. It is used in open-source GPU drivers including i915 for the Intel Integrated GPUs, AMDGPU for the AMD GPUs, and Nouveau for the NVIDIA GPUs.

We describe the design and implementation of GPUvm based on the Xen hypervisor [11]. We develop several optimization techniques to reduce the overhead in each GPU virtualization. Our experiments show that GPUvm incurs different overheads as the level of the exposed interfaces is changed. The results also show that a coarse-grained fairness on the GPU among multiple VMs can be achieved using GPU scheduling.

We also present GLoop, a runtime system to consolidate GPGPU applications including GPU eaters. GLoop is an application-assisted cooperative resource virtualization. While GLoop requires modifications of applications, GLoop can schedule GPU eaters, which can monopolize a shared GPU with the hypervisor-level GPU virtualization approaches. GLoop introduces an event-driven programming model into GPUs, which is widely used in cloud applications driven by I/O events such as network packet arrival [71]. The event-driven programming model allows GPU eaters to be consolidated without wasting GPU time, while GLoop schedules them on a shared GPU in an isolated manner. In addition to consolidating I/O-driven GPU eaters, GLoop allows compute-intensive GPU eaters written in the event-driven programming model to exploit the idle-time of an underutilized GPU. The GLoop runtime also schedules GPGPU applications on the basis of a proportional share scheduling policy.

We prototype GLoop on an unmodified proprietary NVIDIA driver and CUDA SDK. We port eight GPU eaters on GLoop: TPACF, LavaMD, MUMmerGPU, Hybridsort, Grep, Approximate Image Matching, Echo Server, and Matrix Multiplication Server. We perform an experimental evaluation of our prototype demonstrating that our GLoop-based applications are comparable in performance to the original versions and that GLoop successfully consolidates and schedules them on the basis of the scheduling policy. We also show that GLoop's consolidation contributes to improving GPU utilization in two consolidation scenarios: GPU server consolidation and GPU idle-time exploitation.

This dissertation makes following contributions.

- The design, implementation, and evaluation of the hypervisor-level GPU virtualization approaches clarify the tradeoffs and technical difficulties in the virtualization approaches. The clarified tradeoffs between performance, guest device driver modification, and software stack limitation allow cloud software developers to select efficient virtualization solutions for the specific use of GPUs. The detailed analysis identifies the bottleneck of the full-virtualization approach. The evaluation result shows that the full-virtualization incurs non-trivial overhead and implies that the nested page table support in GPUs is promising hardware extension for the full-virtualization approach.

- The application-assisted approach for GPU resource virtualization achieves time-multiplexing of accelerators even without the support of preemption. The case studies for eight GPU eaters show an event-driven programming model in GPUs is applicable. The result envisions our approach can be applied to non-preemptive accelerators such as low-end GPUs and FPGAs.
- GPUvm, PVDRM, and GLoop are provided as a complete open-source software at `https://github.com/CPFL/gxen`, `https://github.com/CPFL/pvdrm`, and `https://github.com/CPFL/gloop`.

1.4 Organization

This dissertation is organized as follows. Chapter 2 describes the model of the discrete GPUs that are focused in this dissertation. Chapter 3 describes existing approaches for resource virtualization of GPUs in detail. Discussion in the chapter motivates sharing GPUs at multi-tenant cloud platforms, and illustrates the missing features and analysis in the field of resource virtualization of GPUs. Chapter 4 shows GPUvm, our GPU virtualization techniques at the hypervisor. The chapter presents the design, implementation, and evaluation of our approaches to show the tradeoffs among the levels of abstractions between GPUs and VMs. Chapter 5 introduces GLoop, an application-assisted approach for scheduling GPU eaters. The chapter shows GLoop's mechanism for scheduling GPU eaters with the co-operation of the applications, and demonstrates that GLoop achieves its goal by the evaluations including realistic scenarios. Chapter 6 concludes this dissertation and discusses the future directions.

Chapter 2

GPU Model

We describe the model of the discrete GPUs, which are focused in this dissertation. The system is composed of a multi-core CPU and a GPU connected on the bus. A compute-intensive function offloaded from the CPU to the GPU is called a *GPU kernel*, which can produce a large number of compute threads running on a massive set of compute cores integrated in the GPU. The given workload may also launch multiple kernels within a single process.

GPU has hierarchical parallelism. Compute threads are grouped into a *warp*, where threads run in lock-step. Warps are grouped into a *thread block*, which runs on the same processor core, called a *streaming multiprocessor* (SM). GPU kernel is a grid that consists of many thread blocks.

The product lines of the GPU vendors are closely tied to the programming languages and architectures. For example, NVIDIA invented the Compute Unified Device Architecture (CUDA) for use as a GPU programming framework. CUDA was first introduced in the Tesla architecture [56], followed by the Fermi and Kepler and later architectures [56, 57]. The GPUvm and GLoop prototypes presented in this dissertation assume these NVIDIA technologies, yet its design concept is applicable for other architectures and programming languages.

Figure 2.1 illustrates the resource management model of our target GPU, which is well aligned with, but is not limited to, the NVIDIA architectures. The detailed hardware mechanism is not identical among the different vendors, although recent GPUs have adopted the same high-level design.

Memory-mapped I/O (MMIO): The current GPU form is an independent

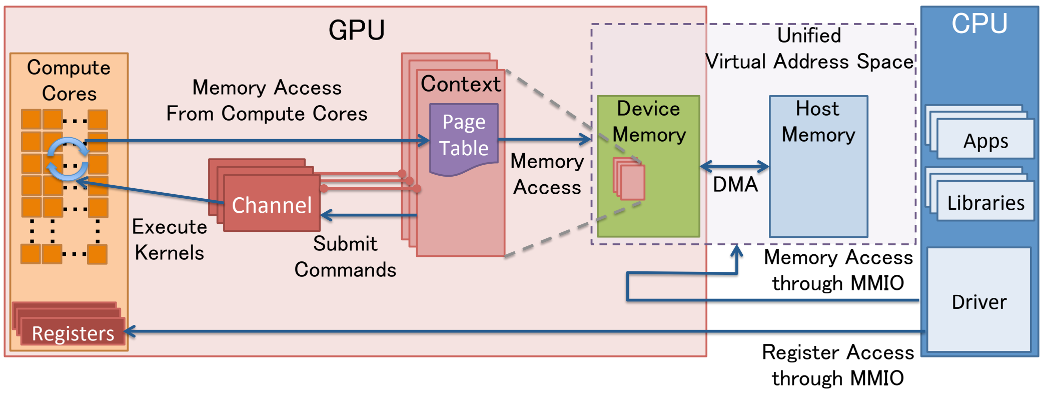


Figure 2.1: GPU resource management model.

computing device. Therefore, the CPU communicates with the GPU via MMIO. MMIO is the main interface that the CPU uses to directly access the GPU, while the hardware engines for the direct memory access (DMA) are supported for transferring large amounts of data. We must note that the I/O ports are used to indirectly access the above MMIO regions. The I/O port is rarely used since it is intended to be used in the real mode, which cannot map a high memory address. In fact, Nouveau, which is an open-source device driver, never accesses it.

GPU Context: Just like the CPU, we must create a context to run on the GPU. The context represents the state of the GPU computing, part of which is managed by the device driver, and owns a virtual address space in the GPU. Multiple active contexts can coexist on the discrete GPU.

GPU Channel: Any operation on the GPU is driven by commands (e.g., launching a kernel) issued from the CPU. This command stream is submitted to a hardware unit called a GPU *channel* and is isolated from the other streams. A GPU channel is associated with exactly one GPU context, while each GPU context can have one or more GPU channels. Each GPU context contains GPU channel descriptors for the associated hardware channels, each of which is created as a memory object in the GPU memory. Each GPU channel descriptor stores the settings of the corresponding channel, which includes a *page table*. The commands submitted to a GPU channel are executed in the GPU *compute cores* and the execution is confined to within the associated GPU context. For each GPU channel, a dedicated command buffer is allocated in the GPU memory that is visible to the

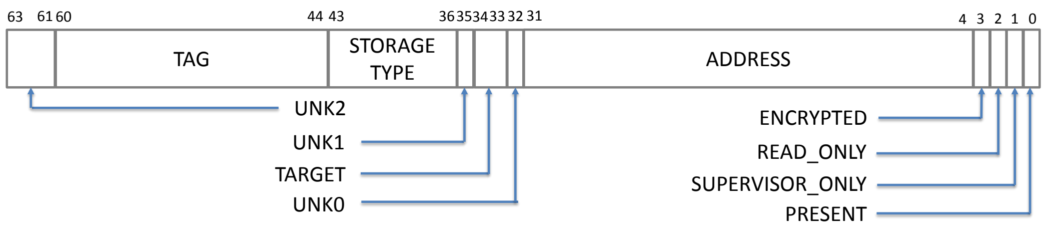


Figure 2.2: Format of GPU page table entry.

CPU through MMIO. The GPU commands can be simultaneously submitted from multiple GPU contexts through the GPU channels. The GPU context switching and command executions in the GPU compute cores are scheduled internally by the GPU hardware.

GPU Page Table: Paging is supported by the GPU. The GPU context is assigned using the GPU page table, which isolates the virtual address space from the others. The GPU page table is separated from the CPU page table. It resides in the GPU memory and its physical address is in a GPU channel descriptor. All the commands and programs submitted through the channel are executed in the corresponding GPU virtual address space.

The GPU page tables translate a GPU virtual address into not only a GPU device physical address but also a host physical address. Figure 2.2 shows the format of the page table entries in the NVIDIA Fermi architecture [49]. `TARGET` indicates the memory type of the target page. We specify a memory type among the following three types; `VRAM`, `SYSRAM` and `SYSRAM_NO_SNOOP`. When the `TARGET` entry is `VRAM`, the GPU page table translates a given GPU virtual address to a GPU device physical address. When the `TARGET` entry is `SYSRAM` or `SYSRAM_NO_SNOOP`, the GPU page table translates a given GPU virtual address to a host physical address. This enables the GPU page table to unify the GPU memory and host main memory into the unified GPU virtual address space. The commands executed in the GPU context can access the host physical memory using the GPU virtual address by leveraging the GPU page tables.

The GPU context uses a GPU virtual address that indicates the host physical address in the GPU page table for initiating the DMA to the associated host memory.

PCIe BAR: The host computer is based on the x86 chipset and is connected to the GPU on the PCI Express (PCIe). The base address registers (BARs) of the PCIe, which work as the windows of MMIO, are configured at the boot time of the GPU. GPU control registers and GPU memory apertures are mapped onto the BARs, allowing the device driver to configure the GPU and access the GPU memory. For example, NVIDIA Quadro 6000 has three BARs, BAR0, BAR1, and BAR3. BAR0 is used as GPU control registers. BAR1 and BAR3 work as GPU memory apertures allowing the device driver to access the GPU memory.

Documentation: GPU vendors currently withhold the details of their GPU architectures due to marketing reasons. Implementations of the device drivers and runtime libraries are also protected by the binary proprietary software, whereas the compiler source code from NVIDIA has recently been open-released to a limited extent. Some previous works have uncovered the black-boxed interaction between the GPU and the driver [54]. The Linux kernel community has recently developed Nouveau [18], which is an open-source device driver for NVIDIA GPUs. Throughout their development, the details of the NVIDIA architectures have been well documented in the Envytools project [49]. Interested readers are encouraged to visit their website.

Chapter 3

Related Work

As GPUs play a critical role in the field of massively data-parallel computing, sharing GPU computing resources gains attention from numerous researchers. The studies spreads on the wide variety of the contexts including multi-tasking of GPU applications in a single desktop machine, sharing GPUs between the resource-containers, and sharing GPUs among VMs in the cloud. This chapter overviews the existing studies and discusses the importance of this dissertation.

Supporting GPGPU applications in the multi-tenant cloud environments requires resource isolation and performance isolation of shared GPUs. Resource isolation is the mandatory requirement since multiple users run GPGPU applications at the same time. Without the strong isolation mechanism, a malicious or buggy GPGPU application can compromise co-located GPGPU applications.

Performance isolation has two levels. GPU kernel or command scheduler can achieve kernel-level coarse-grained performance isolation. However, without preemption support of GPU kernels, long- or infinite-running GPU kernels can monopolize a shared GPU. Application-assisted approaches can achieve time-multiplexing in a fine-grained manner compared to the other approaches.

3.1 Multiplexing GPUs at Operating Systems

As the number of applications using GPUs increases, multiplexing GPUs at the OS layer becomes important. These GPU resource managers work at the OS layers, thus simply bringing these approaches to the cloud using hypervisors are not possible. The OS including GPU drivers achieves resource isolation by using process abstraction: GPU drivers assign isolated GPU memory to the processes. These approaches aim at achieving performance isolation of GPU computing among GPU applications by using GPU commands or kernel scheduling, but these resource managers are of limited use when GPU eaters are executed concurrently on a GPU.

GPU command-based schedulers schedule GPU commands submitted from multiple GPU applications. TimeGraph [43] offers a GPU command-based scheduler that issues GPU commands submitted from processes on the basis of the scheduling policies. TimeGraph inserts GPU commands causing interrupts at the end of submitted group of commands. When the interrupt occurs, TimeGraph wakes up and submits the next group of commands. Since TimeGraph requires GPU driver modifications, it is difficult to run on proprietary software stacks.

Disengaged scheduler [55] schedules GPU commands with a sophisticated probabilistic model without modifying GPU drivers. It first profiles the execution time of each GPU command submitted from each GPU context by trapping accesses to GPU's MMIO region. Then, disengaged scheduler allows GPU contexts to submit GPU commands without traps and approximates the execution time of each GPU context based on the profiled information. This scheduler is efficient since it does not trap MMIO accesses most of the time. Even with these command-based schedulers, a GPU eater can still monopolize a GPU by issuing a command for polling or launching a long-running kernel. To avoid this situation, we have to redesign such applications to issue numerous GPU commands instead of one polling command or split their GPU kernels.

GPU kernel-based schedulers schedule GPU kernels launched from GPGPU applications. Gdev [44] multiplexes a GPU device at the OS level. Gdev introduces a Linux kernel module that makes GPU computing resources accessible from both user and kernel spaces. Gdev has a GPU scheduler whose scheduling points are GPU kernel launches. The Gdev scheduler implements a novel bandwidth-aware non-preemptive device (BAND) scheduling algorithm that extends the Credit scheduler to deal with the non-preemptive and burst nature of GPU applications.

PTask [67], where a GPGPU application is designed as a data flow graph that consists of GPU kernel modules, schedules GPU kernels when they are launched.

These kernel-based schedulers suffer from the same problem as the command-based ones.

3.2 Multiplexing GPUs at Hypervisors

Virtualizing GPU computing capabilities at the hypervisor is a natural way to share GPUs in the cloud using hypervisors. While full-virtualization approach is ideal since it does not require modifications of applications, runtime, and drivers, the existing approaches do not clarify their tradeoffs of performance, isolation, and modifications. Moreover, while these approaches are application-transparent, they cannot schedule GPU eaters.

Amazon EC2 G1 Instance [6] is categorized into I/O pass-through and provides GPU instances. It makes use of the pass-through technology to expose a GPU to an instance. The virtualization extensions of I/O memory management units (IOMMUs) such as Intel VT-d [2] allow devices to be assigned to guest VMs in an isolated manner. Since a pass-throughed GPU is directly managed by the guest OS, we cannot multiplex the GPU on a physical machine.

API remoting, in which the API calls are forwarded from the client to the server that has the GPU, have been widely studied. GViM [31], vCUDA [72], and rCUDA [23] forward CUDA APIs. VOCL [85] does this forwarding for OpenCL. VMGL [50] achieves the API remoting of OpenGL. SnuCL [47] offers OpenCL API remoting backed by heterogeneous CPU/GPU clusters. MultiCL [5] extends SnuCL to support cross-device scheduling of kernel launches by decoupling command queues from specific devices. gVirtuS [26] supports the API remoting of CUDA, OpenCL, and part of OpenGL. In these approaches, the applications are inherently limited to the APIs the wrapper-libraries offer. Keeping the wrapper-libraries compatible to the original ones is not a trivial task because new functionalities are frequently integrated into the GPU libraries, including CUDA and OpenCL. Moreover, API remoting requires that the all the GPU software stacks, including the device drivers and runtimes, become part of the TCB.

VMware SVGA2 [22] para-virtualizes GPUs to mitigate the overhead of virtualizing the GPU graphics features. SVGA2 exposes a virtual GPU device, VMware SVGA2 card, to the guest VM. The para-virtual driver for SVGA2 GPU works in the guest VM and interacts with the host GPU stack through the virtual

GPU. The SVGA2 handles graphics-related requests using the SVGA3D protocol, which is an architecture-independent communication, to efficiently perform 3D rendering and to improve the portability by hiding the physical GPU hardware. This approach is specific to graphic acceleration because SVGA2 targets graphic commands.

Gottschalk et al. proposed a low-overhead GPU virtualization, named LoGV, for GPGPU applications [28]. Their approach is categorized into a para-virtualization where the device drivers in the VMs send requests for resource allocation and mapping memory into the system RAM to the hypervisor. This work exhibits para-virtualization mechanisms to minimize the GPGPU virtualization overhead.

gVirt [82] fully virtualizes the Intel integrated GPUs at the hypervisor. However, gVirt is not designed for the architecture on which multiple active channels can coexist such as NVIDIA discrete GPUs; it is required to switch the render contexts on the driver side. gVirt is tailored to Intel integrated GPUs that use the host memory while discrete GPUs use the device memory. This requires a different design of virtualization since this could pose the different performance overhead and bottleneck. gVirt has to integrate an extension of the specifications for the Intel GPUs into the device driver, thus, requires driver modification. The design of full-virtualization for discrete GPUs and its overhead is still unclear.

3.3 Application-assisted GPU sharing

Numerous researchers have studied how GPGPU applications can be made to become more highly functional to fully utilize GPU capacities [30, 48, 52, 73, 74]. Such GPU applications launch long- or infinite-running GPU kernels. For example, GPUfs [74] exposes file systems APIs to a GPU program to efficiently execute a GPGPU application involving file operations and facilitate its development. GPUnet [48] also provides a socket abstraction and APIs suitable for GPU processing. The persistent threads model [30] launches a maximum-sized grid on a GPU. In this model, thread blocks continuously fetch GPU tasks from work queues to execute them without costly kernel launches. The model is effective for irregular parallel applications such as ray traversal [4]. GPGPU applications performing scientific simulations, sorts, and bioinformatics, typically launch a long-running GPU kernel [52, 73].

Unfortunately, these application designs implicitly assume that only one GPGPU application at a time runs on a GPU. Consolidating these types of app, called GPU eaters, on a shared GPU poses an interesting challenge: How can we effectively share a GPU among GPU eaters in an isolated manner? GPUfs- and GPUnet-based applications poll I/O completion to avoid costly GPU kernel launches so that the other GPU kernels can do nothing until the running kernel finishes. We cannot execute two or more persistent thread applications concurrently since the thread blocks in one application are long- or infinite-running over GPU tasks. The GPU kernels of scientific simulations typically monopolize a GPU for seconds, minutes, or even hours.

A naive software approach to scheduling GPU eaters in an isolated manner is to divide the GPU eater's kernels into smaller GPU kernels by splitting the GPU computations and finishing all the running thread blocks. This approach, called kernel splitting, offers scheduling points to typical GPU schedulers that use GPU kernel launches as scheduling points. However, it degrades the performance of the GPU kernels and incurs non-trivial development costs. Since each GPU kernel has GPU hardware resources, including tremendous numbers of registers and shared memory, their allocations/releases in launches/exits are time-consuming, making the latency of the scheduling points high even if a sequence of split GPU kernels does not need to be descheduled. Moreover, it is difficult to divide a GPU kernel into chunks of an appropriate size to offer timely scheduling opportunities because we cannot exactly know the execution time for each part of the kernel in the development phase. In addition, efficient coordination of multiple kernels requires overlapping communications and computations, which involves significant development effort such as orchestrating the host side processing, the host-device data transfers, and the GPU kernel launches.

The elastic kernel [64] transforms physical thread blocks into logical thread blocks and dispatches them to physical resources. It schedules GPU kernels by adjusting the number and size of logical thread blocks spawned in one launch. EffiSha [16] dispatches logical thread blocks on the basis of the scheduler's decisions. These approaches use the ends of logical thread blocks as scheduling points. Therefore, even with them, a GPU eater with long-running thread blocks can still monopolize a shared GPU.

GPUpIO [88] achieves I/O-driven preemption in GPU applications by instrumenting code with save and restore procedures. Instead of waiting for I/O completions by polling, an inserted procedure saves the state of the executing thread block and finishes it. When the I/O operation is completed, GPUpIO executes another GPU kernel that restores the saved state of the thread block. While GPUpIO is effective for I/O polling-based GPU eaters, long-running kernels such as scientific simulations and persistent threads can still monopolize a shared GPU.

GPUShare [27] schedules GPU kernels by controlling the number of executed thread blocks. When the thread blocks are dispatched, each of them checks whether the execution time of the kernel has exceeded a specified period. If so, the thread block does not start its actual code and finishes early. However, GPUShare fails to achieve fine-grained scheduling for polling-based GPU eaters or GPU kernels whose thread block execution is too long because the thread blocks cannot perform periodic checks.

Multi-process service [60] (MPS), which is also known as context funneling [83], concurrently executes multiple GPU kernels on a GPU. MPS redirects all the streams of the running GPGPU applications to one GPU context in a service process. Thus, the redirected GPU kernels simultaneously run within one GPU context. FLEP [84] is similar to EffiSha, but combines MPS with a thread block scheduler to offer spatial multitasking. The persistent threads approach [30] can schedule GPU kernels requested from GPGPU applications. GPU applications add their GPU tasks to a work queue, and active thread blocks execute GPU tasks in the work queue. Since all GPU tasks in these approaches run in the same GPU virtual address space, a GPU request from a buggy or malicious GPU application can destroy or easily hijack other GPU kernels. This is unacceptable in multi-tenant cloud platforms.

3.4 Multiplexing in GPU Hardware

While traditional GPUs are non-preemptive devices, recent high-end GPUs support the hardware preemption mechanism. NVIDIA Pascal GPUs [61] support compute preemption that allows preemption of GPU kernels at instruction-level granularity. However, current hardware support for GPU kernel preemption is not a perfect solution to performance isolation in the multi-tenant cloud environments.

Since no publicly available information shows the availability of software-level preemption control [84], we have no control over GPGPU applications including GPU eaters to schedule them flexibly. For example, cloud vendors cannot proportionally assign GPU resources to a customer's application on the basis of their payments. Moreover, since GPU hardware is not aware of whether an active GPU eater is polling for I/O completion, the hardware-level scheduler blindly assigns timeslices to the polling GPU eater, leading to wasting GPU time [88]. With control over scheduling GPU eaters, the GPU resource managers would be able to intercept I/O requests of GPU eaters and dispatch other hosted GPGPU applications instead of polling-based blocks.

NVIDIA recently announced its NVIDIA Volta architecture with new mechanisms for MPS, called Volta MPS [62]. It enables multiple GPU kernels to run concurrently with their own GPU address spaces. However, a GPU kernel typically exhausts one type of GPU resource and prevents other GPU kernels from running concurrently [64]. Therefore, Volta MPS's fair-sharing scheduling does not work well in multi-tenant use cases, as described in the white paper [62].

3.5 Summary

While there is numerous work on multiplexing GPUs, the tradeoffs of performance, isolation, and modifications are still unclear due to the lack of design, implementation, and analysis of missing approaches.

Section 3.2 shows the tradeoffs between I/O pass-through, API remoting, para-virtualization, and virtualization for Intel integrated GPUs with partial specifications and driver modifications. However, while full-virtualization is an ideal approach in terms of application modifications, the design, implementation, and performance of full-virtualization is not uncovered. This dissertation explores the design and implementation of full-virtualization for discrete GPUs to clarify the bottleneck caused by the current hardware and tradeoffs between performance and level of interfaces.

Section 3.3 describes the application-assisted approaches to gain finer control of GPGPU applications. While Section 3.1 and Section 3.2 show the existing application-transparent approaches, they cannot schedule the advanced GPGPU applications called GPU eaters, limiting applicability of GPU sharing in the multi-tenant cloud environments. This dissertation shows that application-assisted approach, taking the opposite approach to full-virtualization, can schedule GPU eaters.

Chapter 4

GPU Virtualization at the Hypervisor

The objective of this chapter is to show GPUvm, our GPU virtualization approaches at the hypervisor. As shown in Chapter 3, while full-virtualization is desirable in terms of software modifications, the tradeoffs between full-virtualization and the other approaches are unclear. We present the design and implementation of full-, naive para-, and high-performance para-virtualization approaches to clarify the bottleneck and compare the performance characteristics. The experiments show that GPUvm poses different overheads as the level of the exposed interfaces is changed. Full-virtualization shows significant overhead due to the page table shadowing and MMIO handling. Our para-virtualization approaches eliminate these overheads and make performance close to native one.

4.1 Design

The challenge with GPUvm is to show that the GPU can be virtualized at the hypervisor level. The GPU is a unique and complicated device and its resources (such as memory, channels, and GPU time) must be multiplexed like that in the host computing system. Although the architectural details of a GPU are not well-known, GPUvm virtualizes GPUs by combining the well-established techniques in the CPU, memory, and I/O virtualizations of traditional hypervisors.

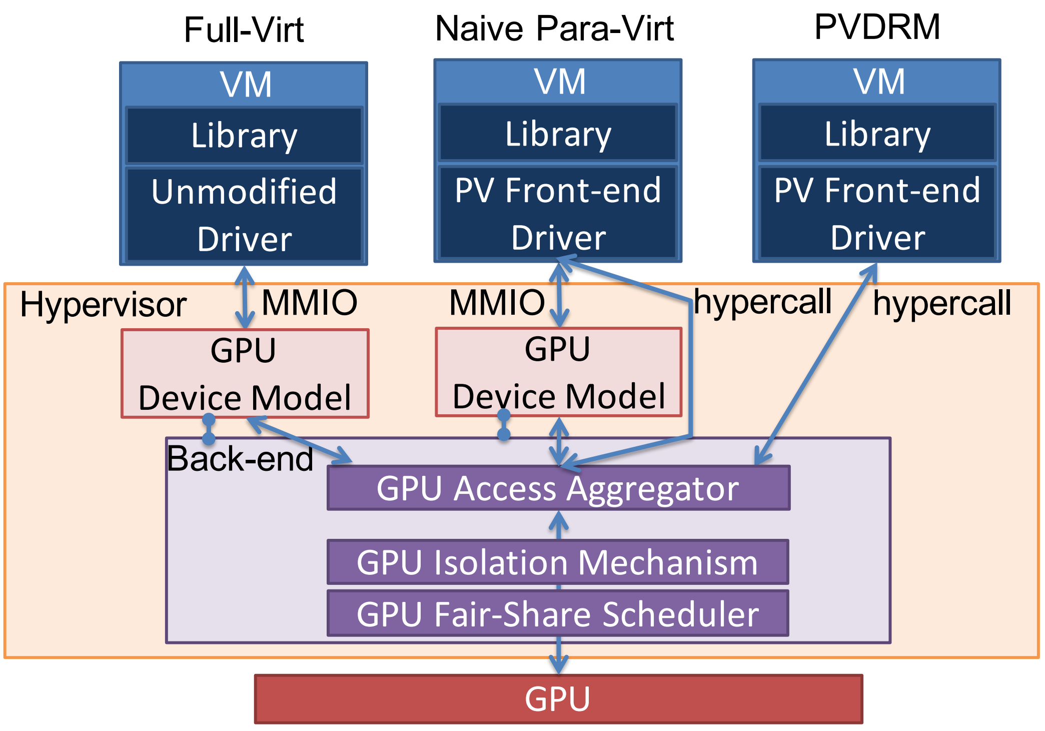


Figure 4.1: Software stack of GPUvm.

4.1.1 Approaches

Figure 4.1 shows a high-level overview of the software stack of GPUvm. GPUvm exposes interfaces to each VM and aggregates the accesses to it. VM operations within the interfaces such as MMIO and hypercalls are redirected to the hypervisor so that the VMs can never directly access the GPU. The GPU Access Aggregator arbitrates the redirected operations to the multiplex GPU resources.

The GPU memory and channels must be multiplexed among multiple VMs to isolate them on the GPU hardware resources. In addition to this spacial multiplexing, the GPU also needs to be scheduled in a fair-share manner. GPUvm logically partitions GPU channels and assigns some of them to each VM. GPUvm also makes use of the GPU page table to isolate the GPU memory among the GPU contexts of different VMs. GPUvm introduces the GPU fair-share scheduler for the GPU command submissions in order to multiplex the GPU computation time.

GPUvm exposes the interfaces on several levels. For the full-virtualization, it exposes a native GPU device model to the VM where the guest device drivers are loaded. For the naive para-virtualization, in addition to the GPU device model, GPUvm provides the interface for updating the GPU page tables. While the full- and naive para-virtualizations use a low-level interface through MMIO, GPUvm exposes the high-level interface for PVDRM, which is the high-performance para-virtualization.

4.1.2 Full-Virtualization

GPUvm supports the hypervisor-level full-virtualization for GPUs., The memory areas, PCIe BARs, and GPU channels must be multiplexed in order to provide an isolated native GPU device model. The main components for the full-virtualization of GPUvm to address this problem include the GPU shadow page tables and GPU shadow channels.

To aggregate the accesses to a GPU device model from a guest device driver, GPUvm intercepts the MMIO by setting these ranges as inaccessible. The accesses to the I/O ports are trapped in the hypervisor, and they are emulated by changing these accesses into ones for the appropriate MMIO region.

In order to ensure that one VM can never access the memory areas of the other VMs, GPUvm creates a GPU *shadow* page table for every GPU channel descriptor. The entire GPU memory address translation is done using the GPU shadow page tables; a virtual address for the GPU memory is translated using the shadow page table not using the one set by the guest device driver. The GPU memory can be safely shared by multiple VMs because GPUvm validates the contents of the shadow page tables. The use of the GPU shadow page tables also guarantees that the DMA initiated from the GPU never accesses memory areas outside those allocated to the VM.

The device driver must establish the corresponding GPU channel to create a GPU context. However, the number of GPU channels is limited in the hardware. GPUvm creates *shadow* channels to multiplex the GPU channels. It configures the shadow channels, assigns dedicated virtual channels to each VM, and maintains the mapping between a virtual channel and shadow channel. GPUvm intercepts and redirects the operations to the corresponding shadow channel when the guest device drivers access the virtual channel assigned by it.

Resource Partitioning

GPUvm partitions the physical memory space into multiple sections of continuous address space, each of which is assigned to an individual VM. The guest device drivers consider that the physical memory space originates at 0, but the actual memory access is shifted by the corresponding size through the shadow page tables created by GPUvm. Similarly, the GPU channels are partitioned into multiple same-sized sections for individual VMs.

The static partitioning is not a critical limitation of GPUvm, and thus, dynamic allocation is possible. When a shadow page table refers to a new page, GPUvm allocates the page, assigns it to a VM, and maintains the mappings between the guest physical GPU pages and the machine physical ones. For ease in implementation, the current GPUvm prototype uses static partitioning. We plan to implement this dynamic allocation in the future.

GPU Shadow Page Table

GPUvm creates GPU shadow page tables in the reserved area of the GPU memory, which translates the guest GPU virtual addresses into GPU device physical or host physical addresses. By design, the device driver needs to flush the TLB caches every time a page table entry is updated. GPUvm can intercept the TLB flush requests because they are issued from the guest device driver through MMIO. After the interception, GPUvm updates the corresponding GPU shadow page table entry.

GPU shadow page tables play an important role in protecting GPUvm itself, the shadow page tables, the shadow channel descriptors, and the GPU contexts from buggy or malicious VMs. GPUvm excludes any memory mappings to the sensitive memory pages from the shadow page tables. Since all the memory accesses by the GPU go through the shadow page tables, no VMs can access these sensitive memory areas.

The current GPU design poses the technical challenge of maintaining consistency between the guest and shadow page tables. In the traditional shadow page tables, page faults are extensively used to detect the updates to the guest page table entries. However, the current NVIDIA GPUs abort the execution of GPU kernels after page faults occur [28,45]. It is impossible to employ the typical shadow page technique that restarts the guest code after setting an appropriate page table entry during page fault handling. Therefore, GPUvm scans all the page tables during a TLB flush.

We must note that GPUvm guarantees the safety of DMA. If a buggy driver sets up an erroneous physical address when initiating the DMA, the memory regions assigned to other VMs or the hypervisor can be destroyed. GPUvm uses shadow page tables and the unified memory model of the GPU to avoid this situation. As explained in Chapter 2, the GPU page tables can map GPU virtual addresses to the physical addresses in the GPU memory and host memory. Unlike in conventional devices, the GPU uses the GPU virtual addresses to initiate the DMA. If the mapped memory happens to be in the host memory, the DMA is initiated. Since the shadow page tables are controlled by GPUvm, the memory access by the DMA is confined in the memory region of the corresponding VM.

GPU Shadow Channel

GPUvm takes the approach of assigning dedicated GPU channels to each VM. As described in Chapter 2, the GPU has multiple GPU channels. They have dedicated command buffers and the driver can simultaneously push commands to the buffers. GPUvm partitions the GPU channels and assigns some of them to each VM. This design enables GPUvm to simultaneously accept GPU commands from the VMs. In addition, compared to multiplexing one GPU channel among the VMs, it does not incur any overhead when switching the GPU context belonging to the GPU channel. Since the GPU commands executed through the assigned GPU channels are confined by the GPU shadow page tables, the isolation among the VMs is maintained.

GPUvm provides GPU shadow channels to isolate the GPU accesses from the VMs. The physical indexes of the GPU channels are hidden from the VMs, but the virtual indexes are assigned to their virtual channels. Mapping between the physical and virtual indexes is managed by GPUvm. GPUvm intercepts the MMIO operations for the virtual GPU channels and then translates the virtual GPU channel indexes into shadow ones and performs the operations to the corresponding channel.

GPUvm provides virtual channel registers to each VM and maintains the map-

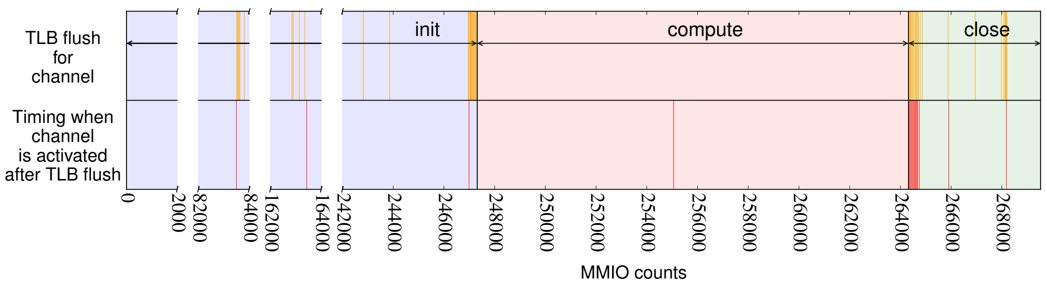


Figure 4.2: Timings of TLB flush and channel activation in GPGPU application (*srad*). The orange lines in this figure represent the TLB flush timing and the red ones are the timings of the channel activations, where the corresponding shadow page table starts to be used.

ping between the physical and virtual channel registers. Since the virtual channel registers are mapped to the memory aperture, GPUvm can intercept any access to them and redirect it to the physical channel registers. Furthermore, intercepting the command submission requests enables GPUvm to schedule the GPU command executions.

GPUvm also creates a GPU *shadow* channel descriptor for each GPU shadow channel to achieve the isolation between virtual GPU channels. The GPU shadow channel descriptors are set for each GPU shadow channel and have a reference to the GPU shadow page table used by the corresponding channel. The GPU shadow channel descriptors reside in the reserved GPU memory and are protected from the VMs in a similar way as for GPU shadow page tables. GPUvm intercepts the GPU memory accesses through MMIO, detects the accesses to the guest channel descriptors, and maintains a consistency between the guest channel descriptors and the shadow ones.

Optimization Techniques

Several optimization techniques are introduced to reduce the overhead in GPUvm.

Lazy Shadowing: In principle, GPUvm reflects the updates of the guest page tables to the shadow page tables every TLB flush. As explained in Section 4.1.2, GPUvm scans the entire page table to find the updated entries in the guest page table. Since TLB flushes frequently occur and the page table size is large, the cost

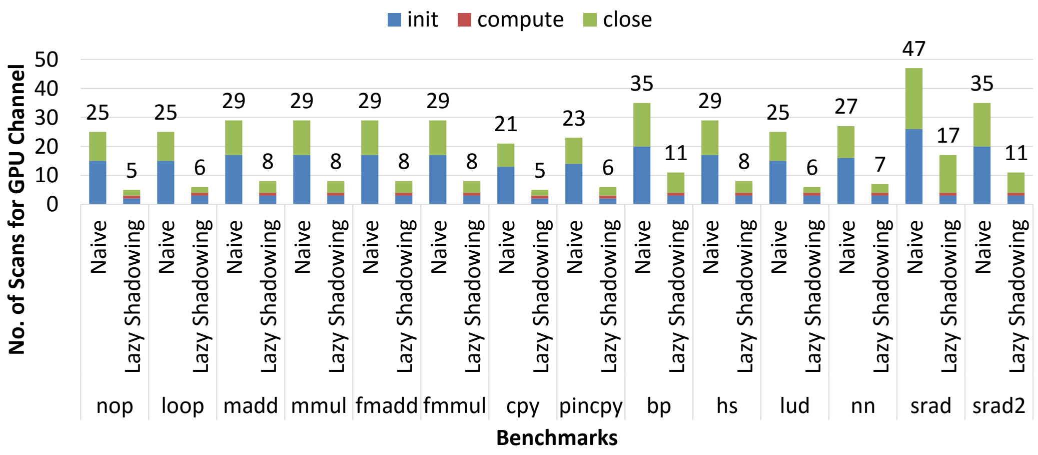


Figure 4.3: No. of scanning shadow page tables in several benchmarks with/without Lazy Shadowing. The measurement is done in each execution phase: *init*, *compute*, and *close*.

of scanning the page tables is significant. Figure 4.2 shows the timings of the TLB flushes and channel activations during the execution of a GPGPU application. Although the TLB is frequently flushed, the shadow page table is often unused immediately after it.

GPUvm lazily scans the guest page tables to reduce the frequency of the scans. It scans the guest page tables each GPU channel activation, which is the timing for using the GPU page table. GPUvm detects the activation by checking the intercepted MMIO operations. Figure 4.3 shows the number of guest page table scans in the benchmarks summarized in Table 4.2. Lazy Shadowing reduces the page table scans in all the benchmarks. Since some of the scans in the *init* phase are delayed until the channel activation point, the scan happens in the *compute* phase.

BAR Remap: GPUvm intercepts the data accesses through the BARs to virtualize the GPU channel descriptors. By intercepting all the data accesses, it maintains the consistency between the shadow GPU channel descriptors and guest GPU channel descriptors. However, this design incurs non-trivial overhead because the hypervisor is invoked every time the BAR is accessed. Figure 4.4 shows the number of BAR accesses in the benchmarks. Even simple benchmarks such as madd significantly access BAR3, causing overhead from the MMIO trappings.

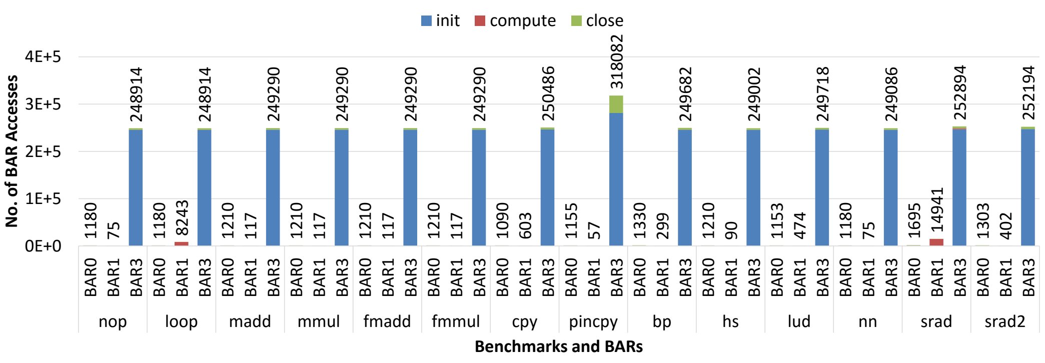


Figure 4.4: No. of BAR0, 1, 3 accesses in benchmarks. The measurement is done in each execution phase: *init*, *compute*, and *close*

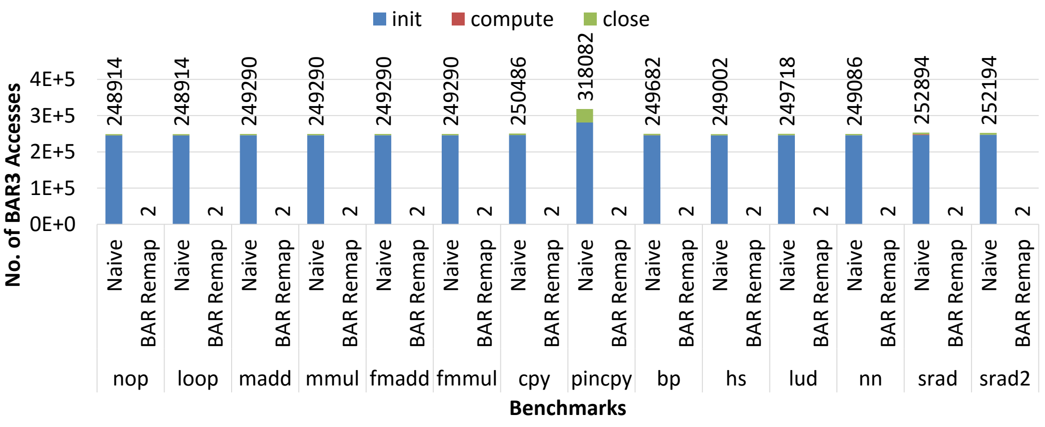


Figure 4.5: No. of BAR3 traps in benchmarks with/without BAR Remap. The measurement is done in each execution phase: *init*, *compute* and *close*

In the BAR Remap optimization, GPUvm passes through the BAR3 accesses other than those for the GPU channel descriptors. Specifically, GPUvm logically partitions the BAR3 area because BAR3 is highly accessed and used for a memory aperture while BAR0 is a control register region and sensitive to virtualization. It exposes part of them as virtual BARs for each VM, and then, GPUvm creates a shadow page table for the physical BAR3. All the accesses to the BAR areas are isolated among the VMs by setting up shadow page tables in the same way as the shadow channels. The number of trapped BAR3 accesses under this optimization is shown in Figure 4.5. This optimization significantly reduces the BAR3 traps in all the benchmarks.

Table 4.1: No. of hypercall issues.

	without multicall	with multicall
nop	11383	118
loop	11383	118
madd	11573	122
mmul	11573	122
fmadd	11573	122
fmmul	11573	122
cpy	12163	105
pincpy	44937	122
bp	11777	137
hs	11429	122
lud	11783	114
nn	11469	118
srad	12379	185
srad2	13031	133

4.1.3 Naive Para-Virtualization

Shadowing the GPU page tables is a major source of overhead in the full-virtualization, because the entire page table needs to be scanned to detect any changes to the guest GPU page tables. We take the naive para-virtualization approach to reduce the cost of detecting the updates. In this approach, we introduce a new hypercall interface for controlling the GPU page tables and integrate it into the full-virtualization mechanisms. The guest GPU page tables are placed within the memory areas under the control of GPUvm and cannot be directly updated by the guest GPU drivers. The guest GPU driver issues a hypercall to the hypervisor to update the guest GPU page tables. The hypervisor validates the correctness of the given page table updates. This approach is inspired by the direct paging in the Xen para-virtualization [11].

We take into account the hypercall invocation cost, which is expensive since the context is switched from the VM to the hypervisor. GPUvm uses the multicall interface that batches multiple hypercalls to reduce any hypercall issues. For example, instead of calling a hypercall to update one page table entry, GPUvm calls one multicall to update multiple page table entries to be updated. Table 4.1 lists the number of hypercalls for each benchmark in the naive para-virtualization with and without the multicall optimization. In all the benchmarks, the multicall dramatically reduces the hypercall issues. Compared to the other benchmarks, pincpy issues many more hypercalls in the naive para-virtualization without the multicall. Pincpy suffers from a larger overhead but the multicall improves its performance, which is described in Section 4.3.1.

4.1.4 PVDRM

While the naive para-virtualization avoids scanning the GPU page tables using a hypercall, it still incurs overhead caused by the low-level interceptions through MMIO and frequent hypercall issues. We also developed PVDRM, the high-performance para-virtualization approach that uses a set of high-level interfaces to address this issue.

PVDRM uses the Direct Rendering Manager (`DRM`) as a boundary of the para-virtualization instead of MMIO. The DRM is a widely used GPU abstraction layer in Linux, and it is used in multiple existing open-source GPU drivers such as in the i915 for the Intel Integrated GPUs, Radeon for the AMD GPUs, and Nouveau for the NVIDIA GPUs. AMDGPU, which is an official driver for the AMD GPUs under development, uses DRM [8]. The use of the DRM provides high-level para-virtualization interfaces and enables for existing software stacks depending on the DRM to work on PVDRM without needing any modification. In addition, since the DRM is used through `ioctls` and each ioctl command semantic rarely changes [9], PVDRM easily maintains the compatibility over driver version changes. In fact, we can interchangeably use Linux kernel v3.6.5 and v3.17.2 on our PVDRM prototype.

PVDRM uses the split driver model [25]. The front-end driver resides in the guest and provides the DRM interfaces to the guest software stack. The DRM operations on the front-end driver are routed to the back-end driver that conceptually runs in the hypervisor. The back-end driver adjusts the routed operations and performs them in the DRM stack.

PVDRM inherits the existing isolation mechanism of DRM to achieve the isolation among the VMs. The DRM has already been integrated with a mechanism that uses the GPU page table to isolate multiple GPU contexts. PVDRM simply uses this isolation mechanism to protect the GPU contexts of different VMs. Shadowing the GPU page tables is unnecessary because the front- and back-end drivers are aware of the GPU virtualization and depend on the DRM isolation mechanism.

4.1.5 GPU Fair-Share Scheduler

So far we have discussed the virtualization of the memory resources and GPU channels for multiple VMs. We herein provide information on the virtualization of the GPU time. This is indeed a scheduling problem. The GPU scheduler of GPUvm is based on the bandwidth-aware non-preemptive device (BAND) scheduling algorithm [44], which was developed for virtual GPU scheduling. The BAND scheduling algorithm is an extension of the CREDIT scheduling algorithm [11] in that (i) the prioritization policy uses a reserved bandwidth and (ii) the scheduler intentionally inserts a certain amount of delay after completion of the GPU kernels, which leads to a fairer utilization of the GPU time among the VMs. Since the current GPUs are not preemptive, GPUvm waits for GPU kernel completion and assigns credits based on the GPU usage. More details about this can be found in [44].

The BAND scheduling algorithm assumes that the total utilization of the virtual GPUs could reach 100%. This is a flaw because there must be some interval in which the CPU executes the GPU scheduler during which the GPU remains idle, causing the utilization of the GPU to be less than 100%. This means that even though the total bandwidth is set to 100%, the credit for the VMs would remain unused, if the GPU scheduler consumes a given amount of time in the corresponding period. The problem is that the amount of credit to be replenished and the period of replenishment are fixed. If the fixed amount of credit is always replenished, after a given period of time all the VMs could have a lot of credit remaining. As a result, the credit may not influence the scheduling decision at all. GPUvm accounts for the CPU time consumed by the GPU scheduler and considers it as the GPU time to overcome this problem. Specifically, GPUvm charges CPU time equally to each VM to avoid an unbalanced charge to a VM that issues short requests frequently.

Note that there is a critical problem in guaranteeing the GPU time fairness. If a malicious or buggy VM starts an infinite computation on the GPU, it can monopolize the GPU time. One possible solution to this problem is to abort the

GPU computation if the GPU time exceeds the pre-defined limit of computation time. Another approach is to cut longer requests into smaller pieces, as shown in [12]. This limitation means GPU eaters can monopolize a GPU due to its long- or infinite-running GPU kernels. Chapter 5 shows that the application-assisted approach can schedule GPU eaters cooperatively.

For future directions, we are planning to incorporate disengaged scheduling [55] on the hypervisor level. The disengaged scheduling provides a fair, safe, and efficient OS-level management of the GPU resources. We believe that GPUvm can incorporate this disengaged scheduling without incurring any technical issues except for the engineering efforts.

4.2 Implementation

Our GPUvm prototype uses Xen 4.2.0, where both domain 0 and domain U adopt the Linux kernel v3.6.5. We target the device model for the NVIDIA GPUs that is based on the Fermi architectures [56]. While the full-virtualization does not require any modification to the guest system software, we make a small modification to the GPU device driver called Nouveau, which is provided as part of the mainline Linux kernel, to implement our naive GPU para-virtualization approach. We implement our PVDRM front- and back-end para-virtualization drivers from scratch.

4.2.1 Full- and Naive Para-Virtualizations

Figure 4.6 shows an overview of our implementation of the GPU device model, the GPUvm back-end, and their interactions with the other components. QEMU-dm is used to create GPU device models and behave as virtual GPUs. The guest device drivers in domain U consider it a normal GPU. It exposes virtual MMIO PCIe BARs to domain Us, trapping the accesses to them. All the accesses to the GPU aggregated by the GPU Access Aggregator are committed to the physical GPU through the `sysfs` interface.

The GPU device model communicates with the GPU Access Aggregator in domain 0, using the POSIX inter-process communication (IPC). The GPU Access Aggregator is a user process in domain 0 that receives requests from the GPU

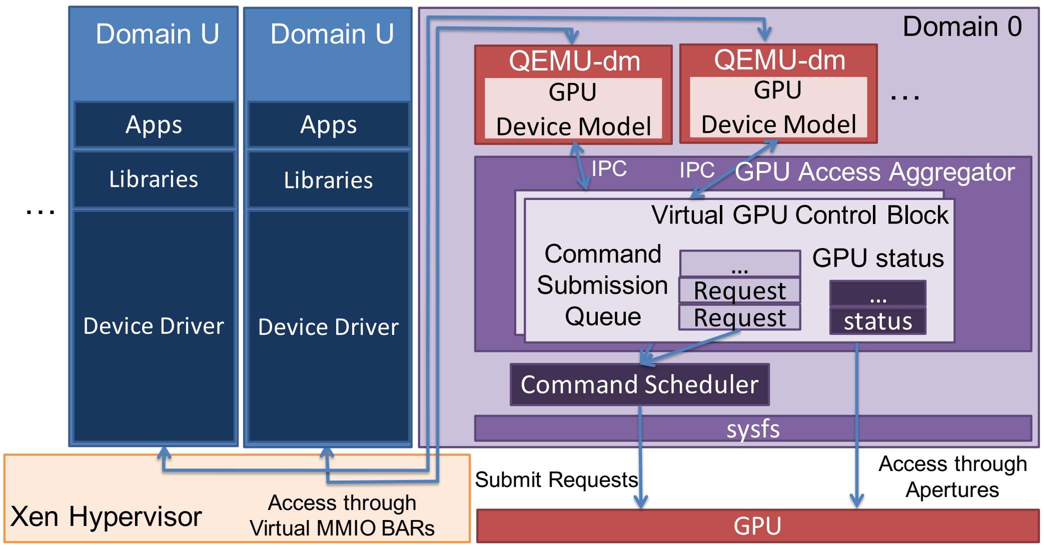


Figure 4.6: Prototype overview of GPUvm full- and naive para-virtualization.

device model, and issues the arbitrated requests to the physical GPU.

The GPU Access Aggregator has virtual GPU control blocks that represent the states of the virtual GPUs. The GPU device models update their own virtual GPU control blocks using the IPC to manage the states of the corresponding virtual GPUs when privileged events such as control register changes are issued from domain U.

Each virtual GPU control block maintains a queue to store the command submission requests issued from the GPU device model. These command submission requests are scheduled to control the GPU executions in the GPU command scheduler. This command scheduling mechanism is similar to TimeGraph [43]. However, the GPU command scheduler of GPUvm differs from that of TimeGraph in that it does not use GPU interrupts. It is very difficult, if not impossible, for the GPU Access Aggregator to insert the interrupt command into the original sequence of commands, because the user contexts may also use some interrupt commands, and the GPU Access Aggregator cannot recognize them once they are fired. Therefore, our prototype implementation uses a thread-based scheduler. Whenever command submission requests are stored in the queue, the scheduler dispatches them to the GPU. Our prototype polls a GPU control register that is modified by the hardware just after the GPU channels become active/inactive to

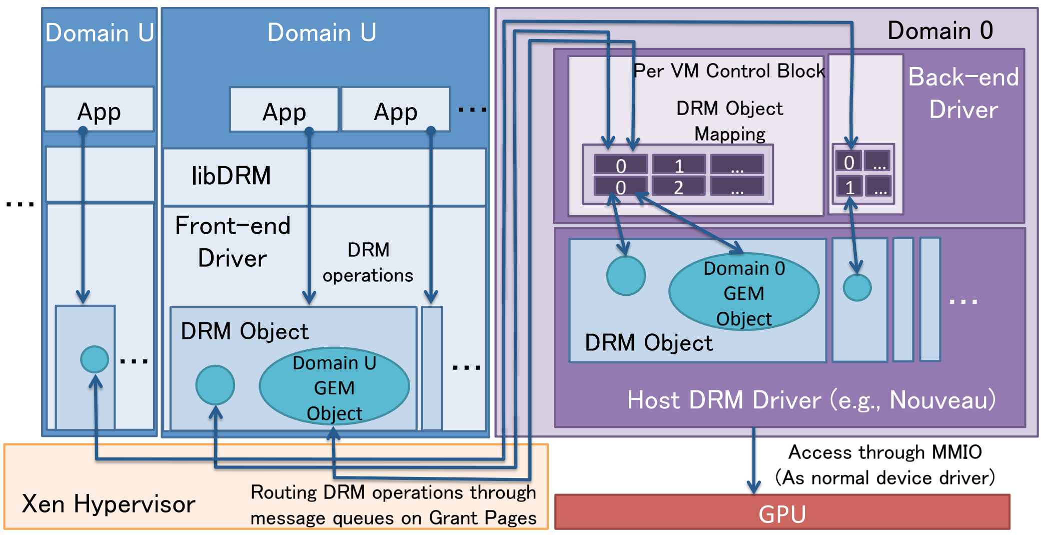


Figure 4.7: Overview of GPUvm PVDRM prototype.

calculate the GPU time.

Another task of the GPU Access Aggregator is to manage the GPU memory and maintain the isolation of multiple VMs in the partitioned memory resources. For this purpose, GPUvm creates shadow page tables and channel descriptors in the reserved area of the GPU memory.

4.2.2 PVDRM

Figure 4.7 overviews our PVDRM prototype. PVDRM front- and back-end drivers are running in the domain U and domain 0, respectively. To access the GPU in the DRM environment, an application creates DRM objects, each of which has its own GPU address space. The DRM object has a memory instance named Graphics Execution Manager (GEM). The application runs its GPU code by sending a request to the GEM object through a special device file. When an application requests to create a DRM object in domain U, the front-end creates a stub DRM object and the back-end prepares the corresponding DRM object. The back-end driver manages the mapping between the stub and corresponding objects. The front-end driver forwards received operations to the back-end, and the default DRM manager running in domain 0 handles the operations. Since each DRM object is isolated, PVDRM guarantees the isolation of each VM DRM object.

PVDRM creates a message queue between the front- and back-end drivers. The queue is used for the front- and back-end drivers to forward the requests and return the operation results, respectively. Both drivers poll a message to efficiently handle the requests and responses.

One technical challenge is to handle a `mmap` operation, which maps a GEM object to the address space of an application. PVDRM needs to map the GEM objects in domain 0 to the address space of the application running in domain U to successfully handle the mmap operation. This means that page sharing across domains is needed. PVDRM shares the pages of the target back-end GEM object with domain U and maps the shared pages to the address space of the application to accomplish this. However, sharing the pages involves several hypercalls, which causes non-trivial overhead. PVDRM pools the allocated shared GEM objects in domain U and domain 0 to mitigate the performance penalty. When the mmap is requested to a cached GEM object, the front-end driver can directly map the object pages to the address space of the application since the cached object pages are already backed to those of the back-end object.

4.2.3 Discussion

GPUvm focuses on the hypervisor-level GPU virtualization for GPGPU, and thus, virtualizes only the GPU resources required for it. This means that GPUvm does not virtualize all of the GPU resources. For example, our prototype does not virtualize a frame buffer used for graphics that is typically used by the X server. Such a resource virtualization is out of the scope of GPUvm.

In shadowing channel descriptors, GPUvm traps the MMIO accesses to the guest channel descriptors to propagate their updates to the shadow ones. The current design of GPUvm implicitly assumes that the updates of the channel descriptors are only done by the CPUs. In other words, GPUvm cannot handle the channel descriptor updates from the GPUs. However, we believe that this assumption is reasonable since the channel descriptors are typically updated by the device drivers. From our experience, we have never seen GPU applications that update the channel descriptors from the GPU contexts.

We must also note the current status of the prototype. Our prototype does not perfectly handle initialization operations. For the ease in implementation, the prototype initializes the physical GPUs by using the domain 0 GPU device driver, and ignores the initialization operations from the guest GPU device drivers just enough to successfully execute the GPU applications in domain U. We need to pay the engineering cost to carefully analyze the initialization operations and sophisticate the corresponding part of the prototype to improve the stability of GPUvm.

We apply the BAR remap optimization to only BAR3 in the prototype, which is dominantly accessed in several benchmarks. In principle, the same optimization can be applied to BAR1 that is used as a memory aperture. This optimization is expected to slightly reduce the overhead in the full- and naive para-virtualizations.

GPUvm does not support those GPUs prior to the NVIDIA Fermi architecture because the previous generations have another way to access GPU memory which is not virtualized in GPUvm. We also restrict our attention to using Nouveau as the guest device driver. The NVIDIA binary drivers should be available with GPUvm, but they cannot be successfully loaded using the current versions of Xen, even in Xen domain 0, which also was the case in the previous work [31, 32].

PVDRM does not have API limitations while API remoting approaches require specific APIs. The DRM interface leveraged by PVDRM allows DRM-based runtimes to work. While the level of this interface is lower than the one of API remoting approaches, it is high enough to abstract components of GPUs. We expect that the performance of PVDRM would be similar to the one of API remoting approaches.

4.3 Experiments

We conduct detailed experiments using a relevant commodity GPU to show the effectiveness of GPUvm. The objective of this section is to answer the following fundamental questions: 1) How much is the overhead of the GPU virtualization incurred by GPUvm?, 2) How does the number of GPU contexts affect the performance?, 3) Can multiple VMs meet the necessary coarse-grained fairness for the GPU resources?, and 4) How much is the overhead of the schedulers we used?. The experiments are conducted on a DELL PowerEdge T320 machine with eight Xeon E5–24700 2.3-GHz processors, 16 GB of memory, and one 2-TB SATA hard disk. We use a NVIDIA Quadro 6000 as the target GPU, which is based on the NVIDIA Fermi architecture. We run our modified Xen 4.2.0, assigning 4 and 1 GB of memory to domain 0 and domain U. Nouveau is running as the GPU device driver and the latest user-mode Gdev [44] is running as the CUDA runtime in domain U. In our previous paper [79], the kernel-mode Gdev was used instead of the user-mode Gdev. The following ten configurations are evaluated: **Native** (non-virtualized Linux 3.6.5), **PT** (pass-through provided by pass-through feature of Xen), **FV Naive** (full-virtualization w/o any optimization techniques), **FV BAR-Remap** (full-virtualization w/ BAR Remap), **FV Lazy** (full-virtualization w/ Lazy Shadowing), **FV Optimized** (full-virtualization w/ BAR Remap and Lazy Shadowing), **PV Naive** (naive para-virtualization w/o multicall), **PV Multicall** (naive para-virtualization w/ multicall), **PVDRM** (GPUvm w/o pooling allocated GEMs), and **PVDRM Pooling** (GPUvm w/ pooling allocated GEMs).

4.3.1 Overhead

We pick several benchmarks from the well-known GPU benchmarks called Rodinia [15] as well as our microbenchmarks to identify the overhead of the GPU virtualization incurred by GPUvm, as listed in Table 4.2. We measure their execution time on the ten models of virtualization. For each benchmark, we run it eleven times, once for warming up and ten times for the results. We use the later ten iterations.

Results

Figure 4.8 shows the average execution times of the benchmarks on each model. The x-axis is the benchmark names while the y-axis exhibits the execution time normalized by one of Native. It is clearly observed that the overhead of the GPU full-virtualization is mostly unacceptable, but our optimization techniques significantly contribute to the reduction of this overhead. The execution times obtained in FV Naive are more than 100 times slower in nine of the benchmarks (nop, loop, madd, fmadd, fmmul, bp, hs, lud, and nn) than those obtained in Native. This overhead can be mitigated by using the BAR Remap and the Lazy Shadowing optimization techniques. Since these optimization techniques are complementary to each other, putting it together improves the performance. The execution time is

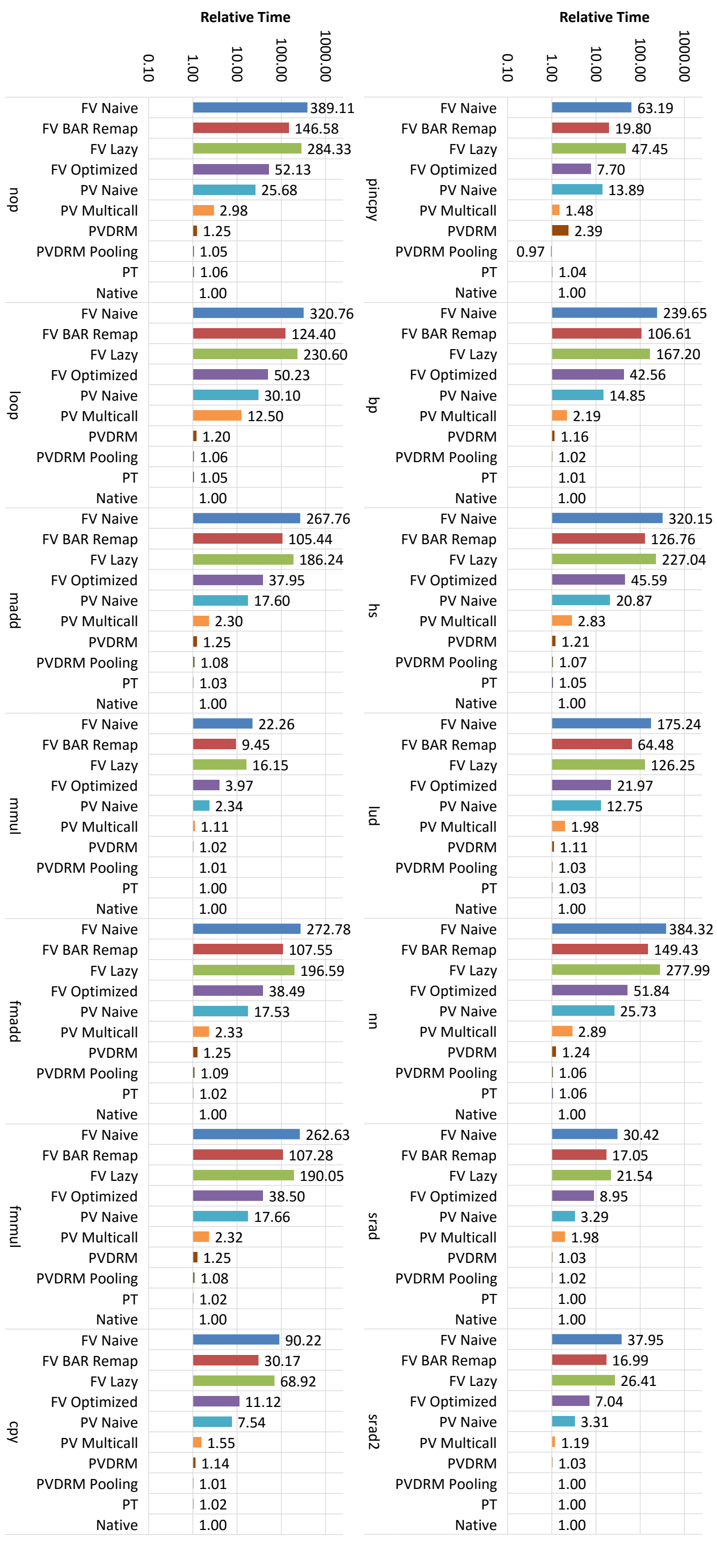


Figure 4.8: Execution time of GPU benchmarks on ten configurations.

Table 4.2: List of GPU benchmarks.

Benchmark	**Description**
NOP	No GPU operation
LOOP	Long-loop compute without data
MADD	1024x1024 matrix addition
MMUL	1024x1024 matrix multiplication
FMADD	1024x1024 matrix floating addition
FMMUL	1024x1024 matrix floating multiplication
CPY	64MB of HtoD and DtoH
PINCPY	CPY using pinned host I/O memory
BP	Back propagation (pattern recognition)
HS	Hotspot (physics simulation)
LUD	LU decomposition (linear algebra)
NN	K-nearest Neighbors (data mining)
SRAD	Speckle reducing anisotropic diffusion (imaging)
SRAD2	SRAD with random pseudo-inputs (imaging)

8.2 times shorter in pincpy (best case) while being 3.4 times shorter in srad (worst case).

We also find from these experimental results that the naive para-virtualization is much faster than the full-virtualization. In most cases, the execution times obtained in PV Naive are 2–20 times slower than those obtained in Native. PV Naive exceeds FV Optimized overall in the execution times. However, in pincpy, it is defeated by FV Optimized. We discuss the reason for this in the next section. The overhead can also be reduced by using our multicall optimization. PV Multicall is at most 3 times slower than Native except in the case of loop.

PVDRM outperforms PV Naive and PV Multicall because the high-level interface reduces the frequency of the MMIO interceptions and hypercalls. In PV-DRM, the overhead becomes 2–25% except for when using pincpy. PVDRM Pooling incurs only a 4% overhead on average. Interestingly, PVDRM Pooling exceeds Native in the performance of pincpy. We discuss this in detail in the next section.

Breakdown

A breakdown of the execution times of the GPU benchmarks is shown in Figure 4.9. We divide the total execution time into five phases: init, htod, launch,

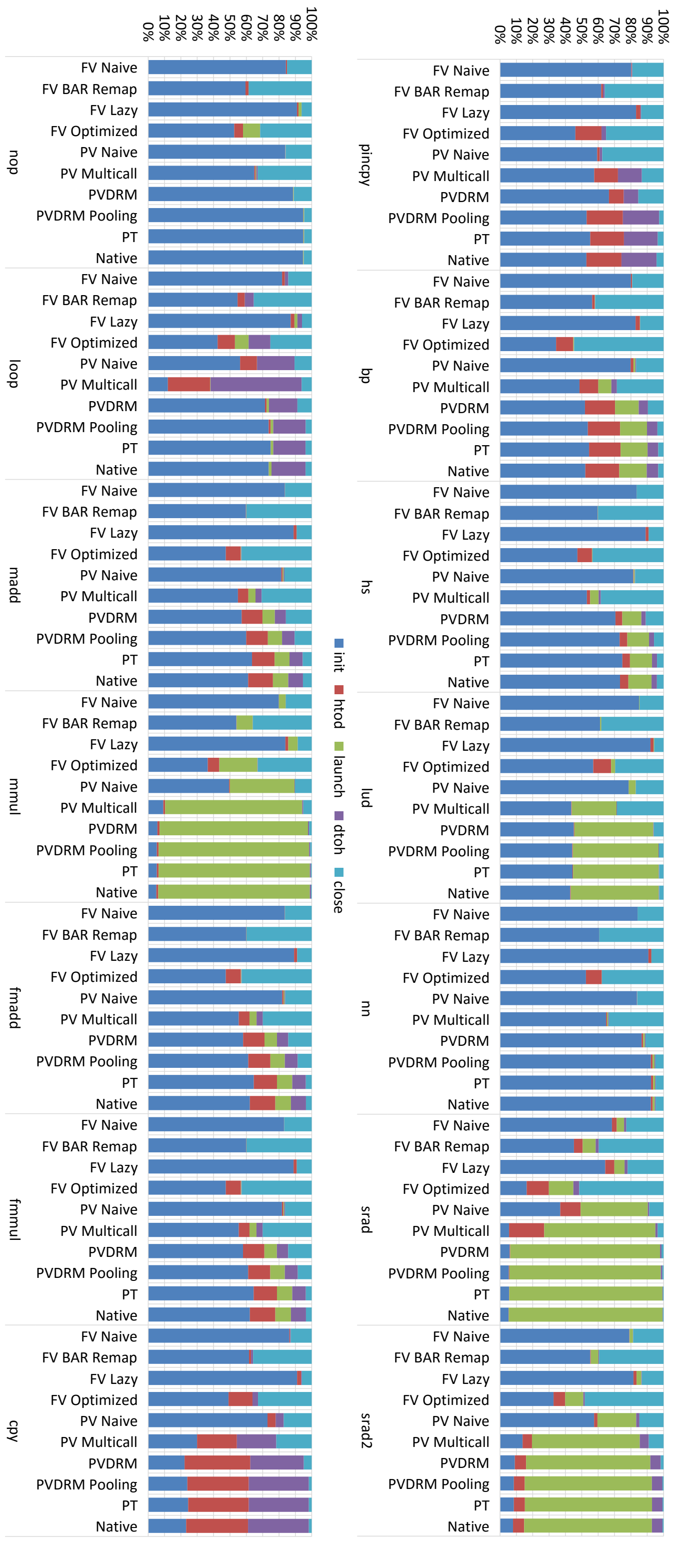


Figure 4.9: Breakdown on execution time of GPU benchmarks.

dtoh, and close. Init is the time necessary for setting up the GPU to execute a GPU kernel. Htod is the time for the host-to-device data transfers, launch is the time for the calculation on the GPUs, dtoh is the device-to-host data transfer time, and close is time for destroying the GPU kernel and context. The figure indicates that the dominant factors in the execution time for GPUvm are the init and close phases. This tendency is significant for four of the GPUvm full-virtualization configurations. In FV Naive, the init and close phases comprise more than 90% of the execution time. The ratios of these phases can be lowered by using optimization techniques and naive para-virtualization. In particular, PV Multicall significantly lowers the ratios of the two phases during computation heavy workloads (mmul, lud, srad, and srad2).

In the case of loop, PV Naive and PV Multicall are more than 15 times slower than Native. Figure 4.9 shows that loop for PV Naive and PV Multicall spends a large amount of time in the htod and dtoh phases. This is because the current GPUvm prototype is not integrated with the BAR Remap for BAR1. Loop transfers a small amount of memory (4KB) back and forth between the host and device. In such cases, the Gdev CUDA runtime uses BAR1 for transferring memory instead of the DMA, and thus, this causes overhead for intercepting the BAR1 accesses.

As explained in Section 4.1.2, BAR Remap and Lazy Shadowing effectively work to reduce the overhead. In all the benchmarks, FV BAR-Remap and FV Lazy incur less overhead than FV Naive. Since both techniques are orthogonal, the use of both optimizations (FV Optimized) results in a bigger gain in performance. On average, the execution times are 2.5 times in FV BAR-Remap, 1.4 times in FV Lazy, and 4.9 times in FV Optimized shorter than in FV Naive.

On the other hand, the init and close phases in two of the GPUvm naive para-virtualized configurations are much shorter than those of the full-virtualization in almost all cases. Full-virtualization performs many operations related to the shadow page tables since memory allocations and deallocations that touch the GPU page tables frequently occur in the two phases. This cost can be significantly reduced in the naive para-virtualizations (PV Naive and PV Multicall) in which those operations are requested by hypercalls. The only exception is pincpy. Pincpy reports larger overhead in PV Naive than one in FV Optimized. This is because pincpy issues many more hypercalls to map a large amount of the host memory in a GPU page table. As shown in Table 4.1, pincpy issues about 4 times more hypercalls than the other benchmarks. Using the multicall effectively reduces this overhead, and pincpy in PV Multicall is 9 times faster than in PV Naive. The multicall optimization reduces the hypercall issues, and as a result, the execution times in PV Multicall are closer to those of PT and Native than the ones of PV Naive. The relative times in six of the benchmarks (mmul, cpy, pincpy, lud, srad, and srad2) in PV Multicall are within twice of ones in Native. In the case of mmul, PV Multicall incurs only 11% overhead.

PVDRM performs comparably to the Native except for pincpy. Figure 4.9 depicts that the init and close times of PVDRM take on a larger ratio than in PV Multicall for pincpy. This is because in the case of pincpy PVDRM issues many hypercalls for granting access to a large amount of the domain 0 memory to the domain U. Pincpy requires that a wide region of the host memory is mmap-ped and it is accessed. Since pincpy involves a mmap operation for a large GEM object, the front- and back-end drivers interact with the hypervisor to map the pages of the back-end GEM object to the guest. PVDRM Pooling uses already-mapped GEM objects, and thus, improves the performance due to less interaction with the hypervisor.

4.3.2 Performance at Scale

We generate GPU workloads and measure their execution times using two scenarios to discern the overhead GPUvm incurs in multiple GPU contexts. In the first scenario, one VM executes multiple GPU tasks, and multiple VMs execute GPU tasks in the other scenario. We first launch 1, 2, 4, and 8 GPU tasks in one VM with the full-virtualized, naive para-virtualized, PVDRM, domain 0, and pass-throughed GPU (FV(Apps-on-1VM), PV(Apps-on-1VM), PVDRM(Apps-on-1VM), Dom0, and PT). These tasks are also run on the native Linux (Native). Next, we prepare 1, 2, 4, and 8 VMs and execute one GPU task on each VM with the full-, naive para-virtualized, and PVDRM(FV, PV, and PVDRM), where all our optimizations are turned on. In each scenario, we run the madd listed in Table 4.2. Specifically, we repeat the GPU kernel execution of madd 10000 times, and measure its execution time.

The results are shown in Figure 4.10. The x-axis is the number of launched

GPU contexts and the y-axis represents the average execution time of application instances in each configuration (e.g. the average execution time of 8 application instances in FV 8 case). This figure shows that the full-virtualization takes longer in the init and close phases as the number of GPU contexts increased. FV also takes longer during these phases as more VMs are running. This is because GPUvm forces the VMs to exclusively access unique GPU resources including the dynamic window.

The graphs also show that the naive para-virtualization and PVDRM of GPUvm have a similar performance to pass-through GPU. The total execution times in PV, PV(Apps-on-1VM), PVDRM, and PVDRM(Apps-on-1VM) are quite similar to those in PT even if there are more GPU contexts.

We can see from this figure that PV (multiple VMs) has a shorter kernel execution time than PV(Apps-on-1VM). This overhead seems to occur for the following two reasons. One is that the device driver uses coarser-grained locks for the GPU resource accesses, and thus, the GPU contexts are executed more sequentially than for one GPU context over multiple VMs; our prototype locks GPU resources in a finer-grained manner. The other is that the MMIO operations of multiple GPU contexts are serialized in the 1VM case since our prototype generates one thread for each VM to emulate the MMIO operations. On the other hand, the ones for a GPU context over several VMs are handled by several threads and are emulated in parallel on physical cores.

The kernel execution time in Native is shorter than Dom0 since Native does not suffer from virtualization overhead caused by Xen.

4.3.3 Performance Isolation

We launch a GPU workload on 2, 4, and 8 VMs and measure the GPU usage on each VM to show how GPUvm achieves the performance isolation among the VMs. For comparison, we use three schedulers: FIFO, CREDIT, and BAND. FIFO issues GPU requests in a first-in/first-out manner. CREDIT schedules GPU requests in a proportional fair-share manner. Specifically, CREDIT reduces the credits assigned to a VM after its GPU requests are executed, chooses a VM whose credit number is positive, and issues its GPU requests. CREDIT reassigns the credits to the VMs for a given interval. BAND is our scheduler that was described

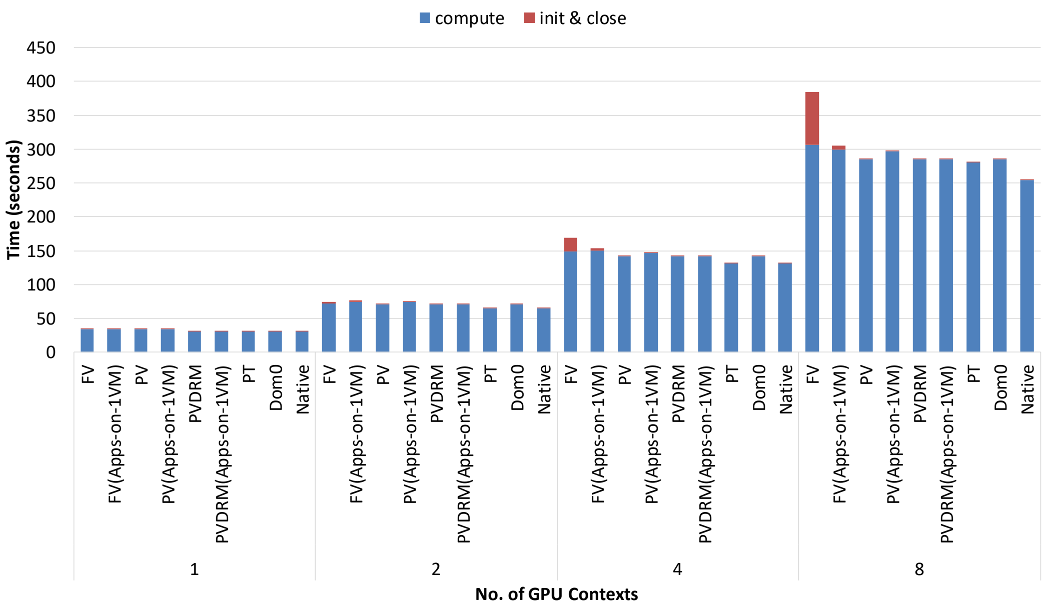


Figure 4.10: Performance across multiple VMs.

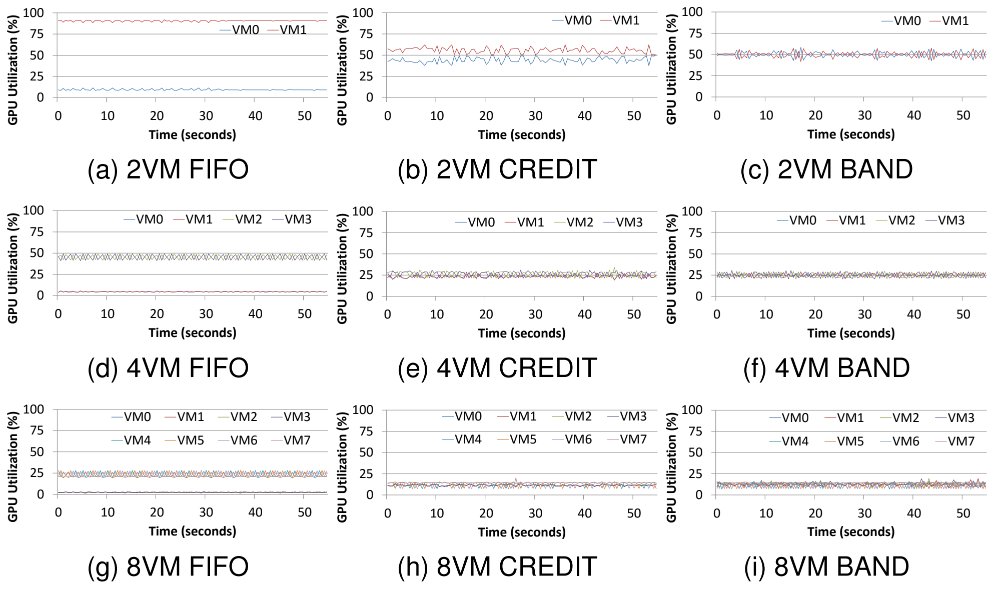


Figure 4.11: GPU usage of VMs (over 500 ms) for three GPU schedulers.

in Section 4.1.5. We prepare two GPU tasks, madd, which is used in the previous experiment and an extended madd (long-madd), which performs 15 times more calculations than the regular one. Each VM loops one of them. We run each task on half of the VMs. For example, madd runs on 2 VMs while long-madd runs on 2 VMs in the 4VM case.

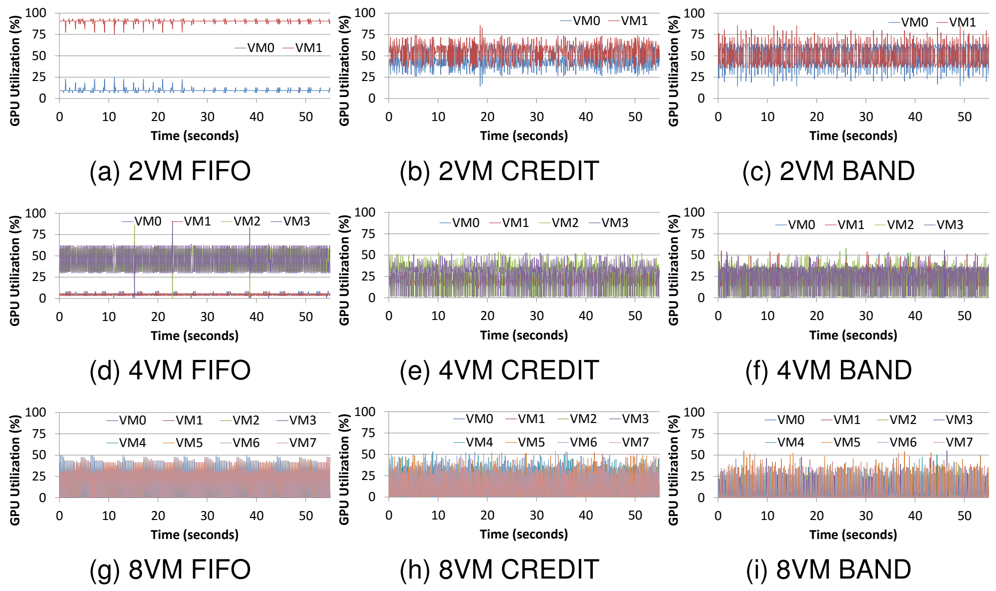
Figure 4.11 shows the results. The x-axis represents the elapsed time and the y-axis is the GPU usage of the VMs over 500 ms. The figure reveals that BAND is the only scheduler that achieves performance isolation in all cases. In FIFO, the GPU usages of the VMs running long-madd are higher in all cases since it dispatches almost the same number of GPU commands from each VM at a given time.

CREDIT fails to achieve the fairness among the VMs in the 2VM case. When the command submission request queue contains requests from only long-madd just after the madd commands have completed, CREDIT dispatches long-madd requests even if it does not have a credit. On the other hand, BAND achieves the fairness because it waits for the request arrivals from madd for a short time period, and thus, handles the requests issued from the VMs whose GPU usage is less.

CREDIT achieves fair-share GPU scheduling in the 4VM and 8VM cases. In these cases, CREDIT has more opportunities to dispatch less-executed VM commands for the following two reasons. First, the GPU operations whose execution times are short are issued more frequently in the 4 and 8VM cases so that there are more points for scheduling VMs in an interval. Last, the request submission queue has requests from two or more VMs just after a GPU kernel completes, which differs from that in the 2VM case.

Note that BAND cannot achieve fairness among the VMs in a fine-grained manner on the current GPUs. Figure 4.12 shows the GPU usages of the VMs over 100 ms. Even with BAND, the GPU usages fluctuated over time, because the GPU is a non-preemptive device. We need a novel mechanism inside the GPU that effectively switches the GPU kernels to achieve a finer-grained GPU fair-share scheduling.

GPUvm achieves coarse-grained fairness by scheduling GPU commands. As described in Section 4.1.5, fine-grained scheduling points are required to schedule GPU eaters effectively since GPU commands include launches of long- or infinite-running GPU kernels. We explore the way to achieve fine-grained fairness by adopting assists from applications in Chapter 5.



(a) 2VM FIFO (b) 2VM CREDIT (c) 2VM BAND

(d) 4VM FIFO (e) 4VM CREDIT (f) 4VM BAND

(g) 8VM FIFO (h) 8VM CREDIT (i) 8VM BAND

Figure 4.12: GPU usage of VMs (over 100 ms) for three GPU schedulers.

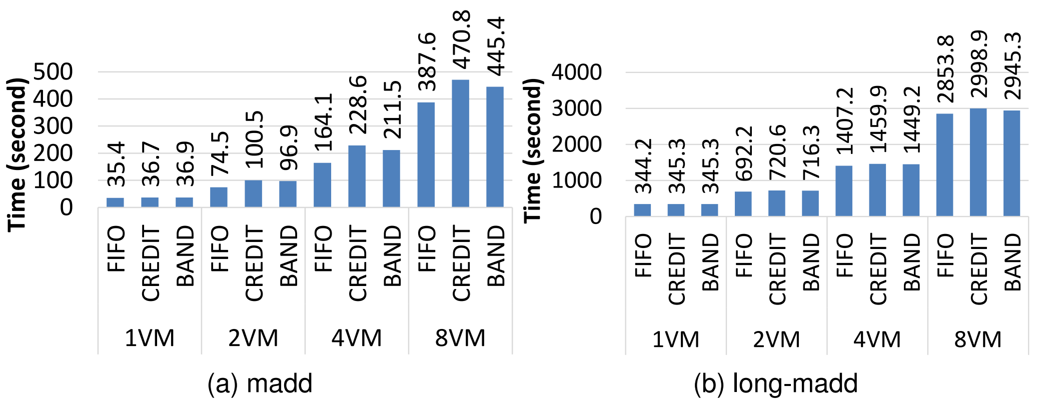


Figure 4.13: Execution time of madd and long-madd with different VMs and scheduling policies.

4.3.4 Scheduling Overhead

We compare the overheads of FIFO, CREDIT, and BAND schedulers, using madd and long-madd benchmarks in the previous section. Madd and long-madd are executed on 1, 2, 4, and 8 VMs, respectively.

Figure 4.13 exhibits the results. The x-axis is the combination of the selected task, the number of VMs and the schedulers. The y-axis is the average of the execution time. The overhead of FIFO is the smallest in our schedulers since FIFO simply issues commands to GPUs just after the commands from madd or long-madd arrived. The execution times of CREDIT and BAND are at most 39% and 30% longer than FIFO in the madd case, while their execution times in the long-madd cases are at most 5% and 3% longer.

In most of the cases, execution time on CREDIT is longer than that on BAND in 2, 4, and 8 VMs. This is because the context switches occur more frequently in CREDIT. Since BAND waits for a small amount of time to receive requests from the previously executed GPU context, BAND tends to execute one GPU context in a batch manner.

Figure 4.13 also indicates that the scheduling overheads are larger in madd than long-madd under the same configuration (the same number of VMs and the same scheduler). Since the madd workload consists of multiple shorter command streams than long-madd, the schedulers run more frequently in the madd cases. In addition to the frequently invoked schedulers, the ratio of the context switches to the whole execution in the madd cases is larger than that in the long-madd cases. Since time for one context switch is constant [65, 81] and madd's execution is much shorter than long-madd, the performance penalty in the madd cases becomes larger.

4.4 Summary

This chapter presented GPUvm, an open architecture for GPU virtualization. GPUvm supports the full- and naive para- and high-performance para-virtualization using optimization techniques. The experimental results using our prototype showed that the full-virtualization incurs a non-trivial overhead largely due to the MMIO handling, and naive para-virtualization provides a two or three times slower performance than the pass-through and native approaches. The high-performance para-virtualization with the high-level interface significantly reduces the overhead.

Table 4.3 summarizes the tradeoffs between the GPUvm hypervisor-level GPU virtualization modes. Although the full-virtualization mode does not require any modification of the software stack on the VMs, its performance penalty is the

Table 4.3: Comparison of GPUvm virtualization modes.

	Full-virtualization	Naive Para-virtualization	PVDRM
Level of Interface	MMIO	MMIO & Hypercalls	Hypercalls
Overhead	297 – 5113%	11 – 1150%	-3 – 9%
Guest Driver Modification	Nothing	Extended for Hypercalls, 540 LOC[1, 2]	Full Scratch, 3474 LOC[2]
Software Stack Limitation	No Limitation	No Limitation	Limited for Linux DRM APIs

[1] Counted additions and modifications related to naive para-virtualization.
[2] Counted by Al Danial's `CLOC`.

highest (297–5113%). The overhead of the naive para-virtualization is lower than that of the full-virtualization at the expense of some modification to the code of the device driver in order to use hypercalls (540 LOC). PVDRM offers the DRM interfaces to the VMs and incurs at most a 9% overhead. The drawbacks of PV-DRM are supporting only applications using the DRM and requiring device driver modification (3474 LOC).

Our suggestion to the GPU hardware design is that a nested page table support in GPUs, similar to Intel EPT, is effective to reduce overhead of GPU virtualization. Since the hardware extension translates guest virtual addresses to machine addresses by setting a physical-to-machine mapping table to a special register in advance, we do not have to scan all page table entries in building the shadow page table. We can offer a virtual GPU by grouping several GPU channels and assigning one nested page table to the group.

For our future directions, the optimization techniques proposed in vIOMMU [10] that can be applied to GPUvm should be investigated. vIOMMU's sidecore IOMMU emulation exposes a shared memory region between the guest and hypervisor as a virtualized MMIO region. Instead of trapping accesses to this region, the hypervisor polls this emulated memory region. This optimization avoids expensive VM-exits caused by traps. We hope our experience with GPUvm gives insight into designing the support for device virtualization such as SR-IOV [21].

Chapter 5

Cooperative GPU Kernel Scheduling

The objective of this chapter is to show the resource virtualization approach for scheduling GPU eaters. While GPU virtualization approaches at hypervisors are application-transparent, they cannot schedule GPU eaters because these approaches schedule GPU commands or kernels. As is mentioned in Section 4.1.5 and 4.3.3, this is because these approaches use the boundary of GPU commands or kernels as a scheduling point while GPU commands or kernels can be long- or infinite-running. This limits applicability of GPU virtualizations in the multi-tenant cloud environments.

This chapter introduces GLoop, which is a software runtime that enables us to consolidate GPGPU applications including GPU eaters. GLoop uses the application-assisted approach. GLoop offers an event-driven programming model, which allows GLoop-based applications to inherit the GPU eaters' high functionality while proportionally scheduling them on a shared GPU in an isolated manner. GPU eaters are modified to be executed on GLoop framework so that GLoop runtime inserts lightweight scheduling points to make GPU eaters schedulable. We carefully designed GLoop to overcome the limitations of existing GPU resource managers. GLoop has three goals, as follows.

- **Consolidates GPU eaters efficiently**: GLoop concurrently executes GPGPU applications on a shared GPU. It dispatches them according to a scheduling policy and lowers scheduling point latency.
- **Provides GPU resource isolation:** GLoop isolates GPU kernel execution.

A malicious or buggy GPGPU application cannot destroy GLoop or other GPGPU applications' contexts.

- **Does not modify proprietary GPGPU stacks:** GLoop works on top of proprietary GPGPU device drivers, GPGPU libraries, and GPU hardware. The current prototype runs on an unmodified NVIDIA device driver and CUDA SDK 9.0.

Our GLoop design is also based on the discrete (off-chip) GPU model that is widely used for its intensive computational abilities. Although GLoop is portable onto integrated (on-chip) GPUs such as Intel GPUs and AMD Kaveri [7], our optimization techniques would not work as well since integrated GPUs have different performance characteristics from those of discrete GPUs. In this case, alternative mechanisms are needed.

We present a prototype of GLoop and port eight GPU eaters on it. The experimental results demonstrate that our prototype successfully schedules the consolidated GPGPU applications on the basis of its scheduling policy and isolates resources among them.

5.1 GLoop Programming Model

An important role of GLoop is to offer low-latency scheduling points to GPU kernels without sacrificing isolation to make GPU eaters schedulable. GLoop provides an event-driven programming model to GPGPU applications by borrowing the idea from Node.js [24]. We treat all kernel operations in this model as events, except for the GPU computation. The events include file I/O, network I/O, and GPU yields. GLoop-based applications register their own callbacks of events of interest, and GLoop dispatches a callback when the corresponding event has completed.

Our programming model has three important features. First, we can develop highly functioning GPGPU kernels. GLoop exposes APIs for event requests such as file I/O, network I/O, and GPU yields; thus, like GPUfs and GPUnet, the development of GLoop-based applications does not involve laborious efforts such as pipelining or asynchronous data copies. Second, we can set scheduling points without splitting GPU kernels or finishing all the running thread blocks. GLoop uses not only GPU kernel launches but also event requests as scheduling points.

```
__device__ void doRead(DeviceLoop* loop,
  uchar* buf, int fd, size_t offset, size_t size){
  fs::read(loop, fd, offset, size, buf,
    [=](DeviceLoop* loop, int read){
      // ...
    });
}
```

Figure 5.1: File Read Program on GLoop.

In addition, the latency of scheduling points decreases since event requests do not involve kernel launches. Third, GLoop's event request APIs are non-blocking. This style of programming allows GLoop to avoid polling-based block behavior and dispatch other hosted GPGPU applications. Namely, I/O operations and GPU computations can be overlapped.

The event-driven programming model is known to be effective in server applications because they are driven by external I/O requests such as network packet arrival [71]. We adopted this model for GPGPU servers in which the GPU applications are driven by events. In addition, this model offers a chance for compute-intensive GPU eaters to exploit the idle resources of an under-utilized GPU. Since the utilization of server GPGPU applications varies, GLoop-based compute-intensive GPU eaters can exploit the idle resources of the GPU. GLoop efficiently schedules compute-intensive GPU eaters with server GPU eaters.

Note that the recent hardware GPU preemption will be complementary to the GLoop technique once preemption technology becomes widespread. When GLoop fails to offer appropriate scheduling points, we can fall back on the GPU preemption to prevent GPU eaters from monopolizing GPUs. The current prototype of GLoop kills GPU applications that monopolize a shared GPU.

5.1.1 Event-driven Programming

In our programming model, GPU kernels of GLoop-based applications are composed of callbacks, each of which is associated with events such as an I/O operation (e.g. file read and write) and GPU yield. When an event is completed, GLoop executes the corresponding callback and unregisters it. The GLoop-based application does not finish until all the registered callbacks have been consumed. A typical GLoop-based application starts and then registers a callback. When the

```
 1: __device__ void doRead(DeviceLoop* loop,
 2:   uchar* buf, int fd, size_t offset, size_t size){
 3:   if (offset < size){
 4:     size_t sizeToRead = min(PAGE_SIZE, size-offset);
 5:     auto callback = [=](DeviceLoop* loop, int read){
 6:       // ...
 7:       doRead(loop, buf, fd, offset + PAGE_SIZE * gridDim.x, size);
 8:     };
 9:     fs::read(loop, fd, offset, sizeToRead, buf, callback);
10:     return;
11:   }
12:   fs::close(loop, fd, [=](DeviceLoop* loop, int err){ });
13: }
```

Figure 5.2: Repeated File Read Program on GLoop.

callback is invoked after the corresponding event has completed, it registers a new callback in the running callback.

We first start with a simple program in the GLoop programming model. Figure 5.1 shows an example program where thread blocks read a file using GLoop APIs. Supported APIs in our prototype are summarized in Appendix A. The function, doRead, reads a specified file size bytes into buf. The program executes fs::read(...), which requests a file read from the host and registers the passed C++ lambda as a callback defined in line 4–6. GLoop executes the registered callback once the requested read has completed. This callback processes read data at line 5.

Figure 5.2 shows another example program where thread blocks repeatedly read a file in a chunking manner using GLoop APIs. The function, doRead, repeatedly reads a specified file up to the specified bytes, size, into buf. We define a callback function, callback, in the C++ lambda style at line 5. This callback processes read data and then calls doRead() again to read a next chunk (lines 5–8). The program executes fs::read(..., callback), which requests a file read from the host and registers the passed callback. As explained, GLoop executes the registered callback once the requested read has completed. Since doRead() is called in the callback, the running callback registers itself again via fs::read() (line 9). These steps are repeated until the read size becomes equal to the specified size. This program represents the example of the loop with GLoop APIs.

In addition, GLoop allows us to register continuation callbacks for GPU yields. This means that we can insert scheduling points into the middle of a thread block execution by posting the continuation as a callback. GLoop supports postTask(), whose argument is the next callback.

5.1.2 Coalesced APIs

GPU execution is based on the hierarchical parallelism of hardware. The GPU kernel is composed of thread blocks. A thread block consists of grouped threads called warps, where the GPU executes threads in lock-step, and this poses the problem of inefficiency when the threads follow divergent paths.

GLoop adopts coalesced API calls, inspired by GPUfs [74] and GPUnet [48]. GLoop specifically forces all the threads in a thread block to call the same APIs with the same arguments at the same point in the application code. The thread block-level approach is reasonable because the GPU offers efficient sharing and synchronization primitives for thread blocks. This means that thread blocks have a coarse-grained parallelism: all the threads in each thread block perform a single task [48, 64]. In addition, managing the GPU kernel at the thread or warp level involves management of much larger metadata per GPU kernel as GPU kernels typically consist of tremendous numbers of threads.

5.1.3 Programming Model Adoption

GLoop programming is a continuation-passing style where each callback represents the next control state. Although we need to modify the application code to consolidate GLoop applications, this is not a complicated task from our experience.

Most GPGPU applications can be made GLoop-based with little effort since there is no need to modify the core logic of the applications. If GPU kernels are short-running, what we should do is to launch the kernels through GLoop. Even if the thread blocks are short-living, there is a concern that numerous short-living thread blocks can occupy a shared GPU. Since GLoop treats thread block completion as scheduling points, as described in Section 5.3.1, no scheduling point insertion is needed in such GPU kernels.

If GPU applications have long- or infinite-running thread blocks, the key to adopting the GLoop programming model is to identify where to insert scheduling points in the target GPU application code. Developers have to pay attention to two types of kernel: (1) I/O-intensive and (2) compute-intensive due to a long-running thread blocks. Regarding the first type, they do not need to insert scheduling points explicitly, since I/O requests are used as scheduling points. Regarding the second, they need to insert a continuation callback into a long-running code path. For example, they set a continuation callback per iteration in a long loop. Therefore, the adoption of our programming model does not involve drastic changes to the application logic.

Inserting callbacks at appropriate points of the code would not impose a huge burden on developers. As a rule of thumb, they should insert continuation callbacks at every location where execution is long. GLoop decreases scheduling point latency by not performing GPU kernel launches in every continuation callbacks. Instead, GLoop only switches kernels if necessary; GLoop dynamically decides to perform context switching at the current scheduling point based on the execution time. If the size of the input for processing becomes larger and thus the kernel execution time becomes longer, GLoop will perform context switching more times. This means that GLoop imposes performance penalties that are small even if a tremendous number of continuation callbacks is set to the application. Automatic insertion of continuation callbacks to appropriate code points is a challenge since it is inherently difficult to obtain such information by statically analyzing the source code. Investigation of this issue is beyond the scope of the dissertation.

By following the above guidance, we successfully implemented eight GPU eaters, as described in Section 5.4.

5.2 GLoop Runtime

GLoop runtime offers an event-driven execution environment which isolates GPGPU applications and requires no modifications to the proprietary GPGPU stack. GLoop forces GLoop-based applications to isolate the execution of GPU kernels in order to establish their own GPU contexts, each of which has its GPU virtual address space [28, 45, 79]. In addition, GLoop mechanisms are on top of existing GPGPU runtime libraries, such as CUDA.

Figure 5.3 shows an overview of GLoop. The GLoop runtime assigns a GLoop-based application a host event loop and device event loops, called a

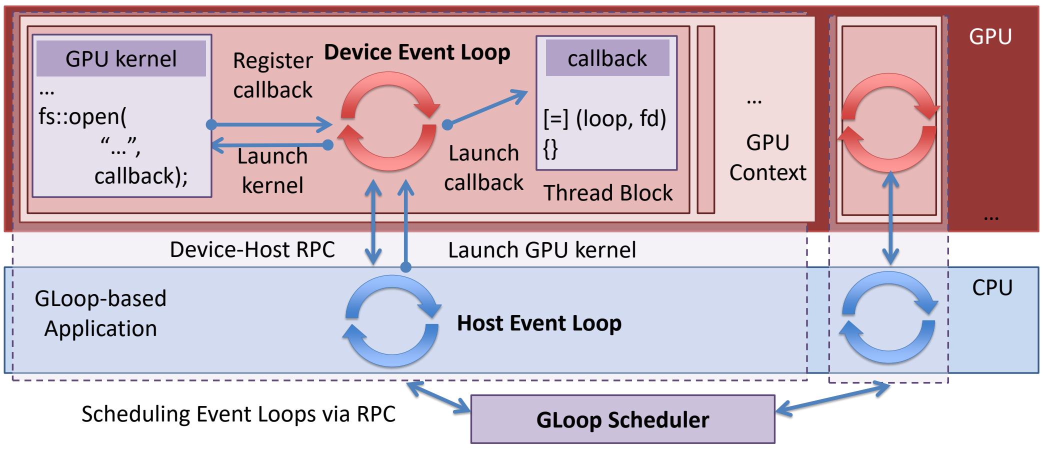


Figure 5.3: Overall architecture of GLoop.

host loop and device loops for simplicity. GLoop scheduler, running as a privileged daemon on a CPU, schedules GLoop-based applications on the basis of its scheduling policy.

Device Event Loop: The device loops, running on a shared GPU, drive the GPU kernel composed of callbacks. They receive an event request from the GPU kernel, pass it to the host loop via the remote procedure call (RPC) slots, and register the callback. This architecture does not involve the GPU kernel finishing/relaunching every event request; device loops can keep running until the suspend signal from the gloop scheduler arrives. When an event has completed, the device loops invoke a corresponding callback.

Note that not all events are pushed to RPC slots. For example, in postTask(), device loops do not access the RPC slots when finding and invoking a registered callback. All the operations are done on the device side, and thus, postTask() offers lightweight scheduling points.

Host Event Loop: The host loop, running on a host CPU, performs events such as I/Os issued by device loops and notifies them of event completion by pushing it to the RPC slots. GLoop-based applications launch GPU kernels through host loop. The host loop communicates with the gloop scheduler to acquire and release the scheduling token, which is the right to use the underlying GPU. When acquiring the token, the host loop launches a GPU kernel that generates device loops.

GLoop Scheduler: The gloop scheduler manages the scheduling token and schedules GLoop-based applications by suspending and resuming the device loops. GLoop sets up a signal slot between the gloop scheduler and device loops to deliver the suspend signal. The signal slot is one type of the RPC slots between the gloop scheduler and device loops. The gloop scheduler accounts the GPU time of every GLoop-based application. In GLoop, GPU time is an interval during which the host loop holds the scheduling token.

5.2.1 RPCs between Host and Device Loop

Device loops interact with the host loop via RPC slots that are on the host-device shared memory. A device loop creates a callback slot in the device memory when requesting an event to the host loop and initializes an RPC slot. The device loop saves the next callback (a lambda function) into the callback slot while writing RPC arguments to the RPC slot. It then issues the RPC by pushing an operation code to the RPC slot. The device loop starts polling the RPC slots for event completion and the signal slot to check whether the suspend signal has arrived.

A host loop polls the RPC slots until the device loops issue an RPC operation. When the RPC operation is detected, the host loop calls a predefined function corresponding to it. After the function has finished, the host loop stores the result and event completion in the RPC slot. The device loops invoke the associated callback when detecting the completion notification.

For example, device loops request the host loop to read a file and associate a callback with the file read's completion. The host loop reads the file, transfers the read data to the device memory through GPU direct memory access (DMA) engines and writes the notification of the read completion to the RPC slot. After the device loop, which is polling the RPC slot, detects the data read by the host loop, it writes the data back to the specified buffer and invokes an associated callback.

5.2.2 Suspend and Resume

Suspending and resuming operations involve all the GLoop components. The host loop requests a scheduling token from the gloop scheduler to start or resume the GPU kernel. The scheduler suspends the currently running device loops by pushing the suspend signal into the signal slot after the device loops have exhausted their timeslice.

The device loops check for the arrival of the suspend signal before invoking a new callback. When detecting a suspend signal, the device loops stop callback invocation and finish execution of the GPU kernel. After the corresponding host loop acknowledges that the device loops have been suspended, the host loop releases the scheduling token to the gloop scheduler and requests it again. The gloop scheduler selects the next host loop to run and passes it the scheduling token. The host loop resumes the GPU kernel by launching a GPU kernel that reconstructs device loops.

Checking the signal slot in device loops is a time-consuming task since the host-device shared memory is accessed through the PCIe bus. This latency in accessing DRAM makes scheduling checks quite slow. This cost is relatively high in postTask() that offers lightweight scheduling points. To reduce this cost, we periodically check the signal slot. The device loops monitor GPU clocks to check for exhaustion of their timeslices (10 ms). The device loops in our prototype access the signal slot every quarter of a timeslice (e.g., 2.5 ms). While the above optimization significantly decreases the number of accesses to DRAM, we can further optimize the latency by placing the signal slot in GPU device memory instead of the host DRAM. This optimization requires the gloop scheduler and GLoop-based applications to share the GPU device memory used for the signal slot, and we note that this is our future work of the GLoop prototype.

5.2.3 An Example of Runtime Execution

We demonstrate how our example 5.2 works with the GLoop runtime. First, the device loop starts doRead function reading a file. If offset is less than size, doRead computes sizeToRead, and invokes fs::read API with the callback. The loop, which is device loop in the code, stores the given callback into the GPU memory, starts RPC for reading a file, and finishes this doRead function execution. The device loop polls the RPC slot since the device loop has the pending slot for the RPC. The host loop receives the RPC from the device loop, performs asynchronous I/O on the host, transfer the data to GPU, and notifies the device loop of the completion of reading the file. The device loop detects the completion of the RPC, load the callback from the GPU memory, and invokes it with the transferred data. Then the callback starts running with the data and does processing.

When the suspension request from gloop scheduler is detected while the device loop polls the RPC slot, the device loop stops polling and finishes the GPU kernel. Since callback is stored in the GPU memory, the device loop can resume the execution after the GPU kernel is rescheduled.

5.3 Design Details

Efficiently consolidating GPU eaters raises three design challenges: (1) how do we control GPU kernels spawning numerous thread blocks, (2) how can we lower scheduling point latency as much as possible, (3) how do we schedule GLoop-based applications in a fair-share manner, (4) how do we build the GLoop runtime on the CUDA-based software stack? To address these challenges, we integrate efficient schemes with the GLoop runtime.

5.3.1 Thread Block Control

The number of thread blocks inside a GPU kernel is critical for scheduling. Suspending all the running thread blocks to de-schedule the GPU kernel is a time-consuming task if the GPU kernel consists of numerous thread blocks. To stop the GPU kernel, a host loop has to wait until the GPU hardware has dispatched all the thread blocks to the SMs. In addition, a GPU kernel generating numerous short-lived thread blocks is difficult to schedule since its code path is too short to insert continuation callbacks. For example, “MUMmerGPU” and “LavaMD” from Rodinia [15] respectively generate up to 65,535 and 125,000 short-lived thread blocks for a single kernel launch.

To address these problems, we introduce a thin thread block scheduler, inspired by the idea of Elastic kernels [64] and EffiSha [16]. Our thread block scheduler, which is a software mechanism running inside the device, puts all the thread blocks in its queue and only executes the same number of thread blocks as concurrently runnable thread blocks on the SMs. The running physical thread blocks fetch logical thread blocks from the queue and execute them. The changes to the application code in order to use this scheduler are trivial: using GLoop's API to retrieve logical thread block information instead of physical ones (e.g. blockIdx and gridDim). The thread block scheduler allows us to complete the suspension of a GPU kernel by only stopping physical thread blocks. When fetching logical thread blocks, the physical thread blocks check for the arrival of a suspend signal. We note that the appropriate number of physical thread blocks relies on resource usage by the thread block. Although we manually specify this number for each GPU kernel on our prototype, an appropriate number can be automatically calculated by using the CUDA occupancy calculator API [58].

5.3.2 Scheduling Point Optimization

To lower the latency of the scheduling points as much as possible, we leverage GPU shared memory regions whose access is faster but whose size is smaller than that of regular device memory regions. We place the control state of the GLoop runtime in a shared memory region.

In addition, GLoop manages two callback slots on the shared memory region. We observe that a GPU kernel typically waits for only one event, which means that it only uses two callback slots. One is for the currently running callback, and the other is for pushing the next callback. We therefore place two callback slots that are currently used in the shared memory region, which leads to quick invoking and saving of callbacks. Although the cost of context switches is logically increased slightly since we need to store the slots from shared memory into the regular memory region, this overhead is negligible.

This optimization can degrade the performance of a GPU kernel if it fully utilizes GPU shared memory. GLoop's shared memory use sometimes results in fewer runnable thread blocks. GLoop can switch the optimization on and off. Developers can thus choose the appropriate GLoop mode for their GPU kernels.

GLoop supports the postTaskIfNecessary() API in order to further lower the latency of scheduling points. This API, used for long loops in the program, allows the GPU kernel to perform lightweight scheduling checks per iteration.

This saves a callback in its argument and returns true only when the device loop checks the suspend signal. Otherwise, it returns false without saving the callback, and the GPU kernel then performs the next iteration. Like postTask(), the API checks the GPU clock to efficiently poll the suspend signal slot. This API allows developers to optimize insertion of lightweight scheduling points at the expense of introducing additional complexity to their code.

5.3.3 Scheduling Policy

We integrate a scheduling policy into the gloop scheduler. An advantage of GLoop over existing GPU resource managers is that it can assign running states to hosted GPU kernels, such as running, ready, and blocked, as in traditional process abstraction because it manages the event invocations of the hosted GPU kernels. We believe that this feature would enable us to integrate various CPU scheduling policies into the gloop scheduler. In particular, the main focus of this dissertation is that GLoop can schedule hosted GPU applications in a fine-grained manner, which is essential for multi-tenant cloud platforms.

Our scheduler proportionally dispatches GPU kernels in a work-conserving manner to fully utilize GPU resources. The scheduler is based on weighted fair queuing [29]. It prepares each user's queue and assigns more GPU time to high priority users by weighting their queues. When a GPU application requests the launch of a GPU kernel, the gloop scheduler pushes its GPU kernel launching request into the application's queue and sets the queue to active if it is inactive. Each queue has a virtual time that elapses during execution of the GPU kernels fetched from the queue. The gloop scheduler selects the active queue whose virtual time is shortest and passes the scheduling token to the host loop. The execution of the GPU application is controlled by GLoop's suspend and resume mechanism explained in Section 5.2.2. When all the GPU kernels in a queue complete, the queue becomes inactive. To achieve a work conserving scheduling, the gloop scheduler adjusts the virtual time of the queue that just becomes active. Specifically, the gloop scheduler resets the virtual time to the shortest virtual time among the active queues.

GLoop's runtime intermediates event invocations and thus assigns the runtime states of the GPU kernels such as I/O waiting, and the gloop scheduler manages the scheduling token assignment on the basis of the runtime states. The gloop scheduler can deschedule GPU applications when all of the thread blocks wait for the events completion. For example, a GPU application is descheduled when all its thread blocks are waiting for newly incoming packets. This feature is effective in the context of consolidation.

5.3.4 CUDA API Scheduling

In designing the GLoop runtime, we take into account the invocation of the CUDA APIs that involve exclusive access to the underlying GPU. While a GPU kernel spawned from another CUDA context is running, some CUDA API call blocks the applications. For example, we cannot initialize a new CUDA context in CUDA 7.5 on Kepler GPUs during the other GPU kernel execution. In this case, the target CUDA context is never initialized until the running GPU kernel finishes. The GLoop runtime needs to suspend the GPU kernel to execute such CUDA APIs.

To address this issue, we reuse the scheduling token used for GPU kernel scheduling. Specifically, the applications acquire the scheduling token from the gloop scheduler to invoke a CUDA API of them. The current design conservatively forces the applications to try to get the token in calling all CUDA APIs. When an application has acquired the scheduling token for the CUDA API invocation, the gloop scheduler suspends the running application, namely stops the running GPU kernel. For example, an application initializes its new CUDA context after acquiring the scheduling token. At this time, the gloop scheduler has already suspended the GPU kernel execution.

5.3.5 Discussion

Large GPU memory transfers between the host and device can also monopolize GPUs. Memory transfers can occupy GPU DMA engines for a long time and block subsequent memory transfer requests. Previous studies [42, 64] suggest splitting large memory transfers into small chunks and scheduling split requests. While GLoop does not focus on memory transfers, these techniques can be integrated into it.

If GLoop-based applications leverage shared memory, their developers need to pay attention to GPU kernel context switching, which clears the shared memory content. There are two ways of addressing this issue at callback boundaries: saving and restoring the content of the shared memory or reconstructing the content. Our ported applications using shared memory can take either way.

Although GLoop provides low-latency scheduling points, the tremendously large number of scheduling points affects overall performance. The scheduling point frequency depends on scheduling point places in the application code. We note that developers can adjust the trade-off between scheduling point frequency and performance penalty by taking into account that the latency is less than 2.26 μs in Kepler and 1.23 μs in Pascal, as shown in Section 5.6.1.

The GLoop architecture is portable to various resource-shared environments. In a container-based system, one possible setup is that the gloop scheduler runs on the host OS as a service process, the host loops in GPU applications use CPU slices assigned to their own containers, and their device loops run on the shared GPU. The gloop scheduler schedules the running GLoop-based applications in the containers. In a VM system where a GPU is virtualized [31, 32, 79, 82], the gloop scheduler is inside the hypervisor or privileged VM, and each loop of the GPU applications runs on virtualized CPUs and the GPU of their VMs.

While GPUs are getting better hardware preemption support, the other types of simple accelerators do not support preemptions (low-end GPUs, FPGAs etc.). The GLoop design can be applied to such accelerators to offer scheduling mechanism in software, and this enables sharing of accelerators in multi-programmed environments without adding complexity to the accelerators themselves.

Since GLoop is a cooperative scheduling mechanism, malicious or buggy GPU eaters can launch GPU kernels aiming at monopolizing a GPU. We categorize such GPU eaters into two types, (1) launching a GPU kernel without taking a token from the gloop scheduler and (2) launching a GPU kernel that does not have GLoop's scheduling points. The former GPU eaters can be detected if GLoop introduces a kernel module that catches launching GPU kernels. Since launching GPU kernels are done by ioctl to a GPU device file or MMIO to a specific region of GPU, a kernel module can detect it by hooking a ioctl or trapping MMIO by protecting memory regions for MMIO [17]. The recent literature [55] demonstrates handling GPU kernel launches by trapping MMIO even with the proprietary NVIDIA GPU driver. This is one of the future extends of the prototype of GLoop.

For the latter GPU eaters, we kill a process of GPGPU applications monopolizing a GPU, that results in stopping GPU kernels. Currently, it is done manually by kill command, but we can extend our prototype to kill automatically when GPGPU application exceeds the threshold.

5.4 Implementation

We implemented a prototype of GLoop on Linux kernel 4.10.0-37 with CUDA 9.0 for NVIDIA Kepler and Pascal GPUs [57,61]. The current prototype is tailored to Linux container-based platforms, which means that GLoop-based applications in each container run on a shared GPU. Figure 5.4 overviews our prototype, which consists of a GLoop library and a gloop scheduler daemon.

We note that GLoop can be prototyped with the existing tool chains such as CUDA. We do not need to prepare special compilers or preprocessing tools for the implementation. This property can improve the GLoop portability that allows us to easily install the runtime and follow the updates of the underlying CUDA runtime.

GLoop Library: The GLoop library and applications are implemented using CUDA, and they are compiled in the NVIDIA CUDA Compiler (NVCC). GLoop-based applications are linked to the library to spawn the host and device loops. Each host loop communicates with the gloop scheduler via the POSIX interprocess communication (IPC). The GLoop-based application invokes GPU kernels through the host loop. To execute GPU kernels, the host loop requests the scheduling token from the gloop scheduler by using the POSIX IPC. When the host loop acquires the token, it starts a GPU kernel that constructs or resumes the device loops on the device side and executes user-written GPU code on the top of them. If the kernel is suspended by the gloop scheduler, the host loop releases the scheduling token to the gloop scheduler and requests the scheduling token again to resume the suspended GPU kernel. This scheduling token request is automatically conducted by the GLoop library.

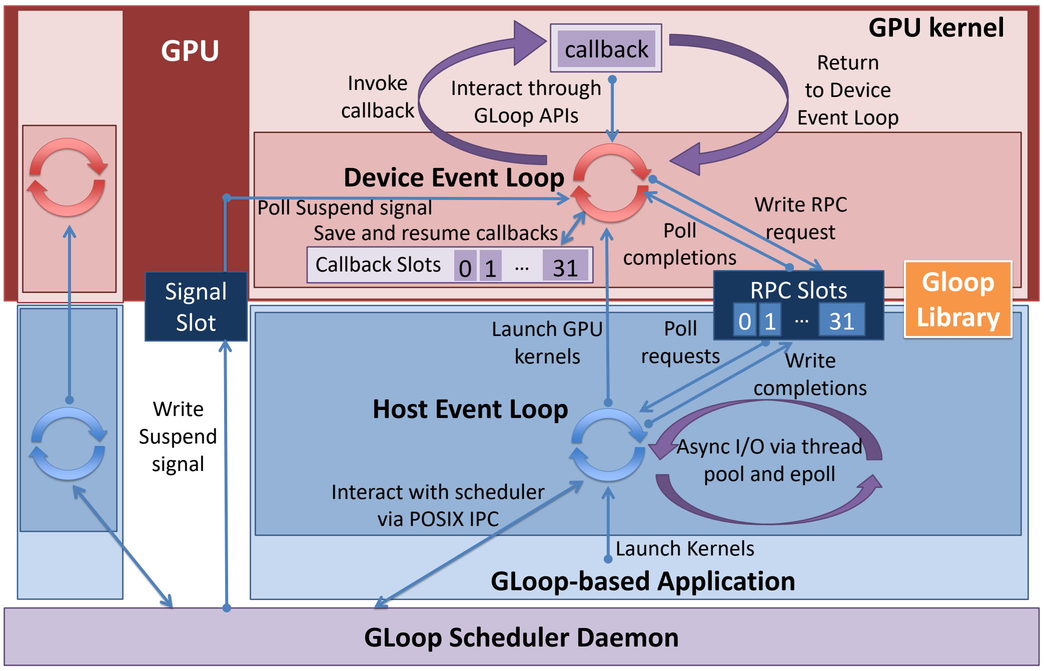


Figure 5.4: Overview of GLoop prototype.

GLoop Scheduler Daemon: The scheduler runs in the host as a daemon and manages the scheduling token. The host loop in each GLoop-based application tries to obtain the scheduling token from the gloop scheduler to execute its GPU kernels. The gloop scheduler sends a suspend signal to the active device loop every timeslice if two or more host loops request the scheduling token. When receiving the suspend signal, the device loops save their state, stop themselves, and finish the GPU kernel. The corresponding host loop releases the token to the gloop scheduler. The gloop scheduler then selects the next host loop to run and sends it the scheduling token.

Callback: We use C++11 lambda supported in the recent NVCC to represent callbacks. C++11 lambda saves data necessary to resume the GPU kernel, i.e., an instruction pointer and captured context data. The NVCC automatically captures variables referred by the lambda and saves them as a lambda object. The device loops use it as the callback.

In NVIDIA GPUs, the PTX ISA, a virtual instruction set for NVIDIA GPUs,

Algorithm 1 Pseudo code of main loop of device loop.

```
function DRAIN
    $slotID \leftarrow Invalid$
    while $slotID$ != $ShouldExit$ do
        if $slotID$ != $Invalid$ then
            $slot \leftarrow Slots[slotID]$
            $data \leftarrow slot.data$
            $callback \leftarrow slot.callback$
            DISPATCH($callback, data$)
            DISPOSE($callback$)
        end if

        if Suspension Request arrives then
            $slotID \leftarrow ShouldExit$
        else if No pending RPC slots exist then
            $slotID \leftarrow ShouldExit$
        else
            $slotID \leftarrow Invalid$
            for each $candidateSlotID \in SlotIDs$ do
                $slot \leftarrow Slots[candidateSlotID]$
                if RPC corresponding to $slot$ is completed then
                    $slotID \leftarrow candidateSlotID$
                    break
                end if
            end for
        end if
    end while
end function
```

version 2.1 introduces *indirect call* [63] that allows CUDA programs to use lambda functions stored in GPU memory. The NVCC intermediately converts CUDA lambdas to C structures holding a pointer to a function and captured variables. Since CUDA lambdas are C structures, we can store and load lambdas in GPU memory. When calling a CUDA lambda, the NVCC emits indirect calls for this function pointer. While our prototype uses indirect calls, GLoop is applicable even without indirect calls, by the complier-level approach collecting the types of lambdas and emitting a large switch statement in a call site.

Algorithm 2 Pseudo code of main loop of host loop.

```
1: function LAUNCH(kernel)
2:     repeat
3:         Acquire a scheduling $token$ from gloop scheduler
4:         LAUNCHONGPU($kernel$)
5:         Release a scheduling $token$
6:     until $kernel$ completes all $callbacks$
7: end function
```

RPC Slots: Due to the lack of atomic operations over the PCIe bus, the device loops and host loop poll their RPC slots and behave in a producer-consumer manner; synchronization is not required since neither will simultaneously read or write to the same slot. The host and device loops issue RPCs in two phases to ensure memory consistency of the shared RPC slots. First, a host or device loop writes RPC arguments and flushes them by issuing a memory fence. The loop then writes and flushes the one word operation code. In checking the slot, the host and device loop bypass CPU and GPU caches, respectively. This protocol guarantees that the arguments are visible when the RPC operation code is detected. This is the similar to the GPU RPC implementation in previous work [74, 77].

The slot holds a callback associated with the given RPC completion. Our prototype stores serialized CUDA lambdas to the memory region allocated for the slots. When the RPC completes, the device loop load the corresponding CUDA lambda from the slot and invoke it.

Host and Device Event Loops: Algorithm 1 shows the pseudo code of the device loops of our prototype. Each device loop in thread block invokes this Drain function that repeatedly drains populated callbacks from the program. If the suspension request arrives or there are no pending RPC requests, device loop finishes their execution. Otherwise, device loop polls RPC slots to retrieve completed RPCs.

Algorithm 2 describes the pseudo code of the main loop of the host loop. The host loop acquires a scheduling token from the gloop scheduler, launches a kernel, and releases a scheduling token when the kernel finishes. The host loop repeats this loop if kernel is suspended by the gloop scheduler. Otherwise, the host loop finishes launching the kernel. Besides the main loop of the host

Algorithm 3 Pseudo code of I/O polling thread of host loop.

```
 1: function IOLoop
 2:     loop
 3:         for each $slotID \in SlotIDs$ do
 4:             $slot \leftarrow Slots[slotID]$
 5:             if RPC request is issued in $slot$ then
 6:                 $data \leftarrow slot.data$
 7:                 Execute asynchronous I/O in thread pool with $data$
 8:                 Following steps are executed when I/O completes
 9:                     Mark $slot$ completed with $data$
10:             end if
11:         end for
12:     end loop
13: end function
```

loop, the host side code runs Algorithm 3 in the different thread. This I/O loop in the host loop repeatedly polls the RPC slots, retrieve the request from the device loops, and perform asynchronous I/O in a thread pool. Once I/O completes, host loop marks slot completed with the data. This completed RPC slot is detected in the device loop.

5.5 Case Studies

GLoop is applicable to various GPGPU applications. We ported eight GPU eaters with different features. Table 5.1 shows the ported GPU eaters and the number of explicitly inserted scheduling points. The number does not include the implicitly inserted scheduling points which are inserted by GLoop runtime at the boundary of logical thread block dispatches.

TPACF: This application from Parboil2 [76] launches a single kernel composed of long-running thread blocks. TPACF calculates the distances between all pairs of astronomical bodies. GLoop splits the kernel to insert scheduling points; we inserted postTask() into the loop calculating the distances of pairs. Note that the original TPACF intensively uses shared memory. Since GLoop switches GPU kernels, the content of the shared memory must be saved and restored every time the kernel is switched. The GLoop version uses regular device memory called

Table 5.1: List of ported GPU eaters.

GPU eater	**# of explicit scheduling points**
TPACF	5
LavaMD	0
MUMmerGPU	2
Hybridsort	2
Grep	19
Approximate Image Matching	13
Echo Server	5
Matrix Multiplication Server	6

global memory instead of shared memory. For comparison, we integrated the same change into the original one in addition to the original TPACF.

LavaMD: LavaMD from Rodinia [15] launches a single kernel that generates many (125,000 in our setting) short-lived thread blocks. Since the thread blocks are short-lived, the kernel cannot be split. Instead, GLoop schedules the logical thread blocks as described in Section 5.3.1. GLoop schedules logical thread blocks on 30 physical thread blocks. Every time the logical thread blocks finish, control returns to GLoop.

MUMmerGPU: MUMmerGPU [70] from Rodinia [15] consists of two long-running kernels (mummergpuKernel and printKernel). The kernels have different features; the former generates 9,766 long-running thread blocks, and the latter generates 65,535 short-lived thread blocks. We inserted postTaskIfNecessary() into the long loop of mummergpuKernel. To obtain more scheduling points, logical thread blocks are scheduled on 30 physical thread blocks in mummergpuKernel. Logical thread blocks in printKernel are scheduled on 60 physical thread blocks.

Hybridsort: Hybridsort [75] from Rodinia launches two kernels: bucket-sort and merge-sort kernels. The bucket-sort kernel generates many short-lived thread blocks, while the merge-sort kernel generates long-running thread blocks whose numbers vary from 8 to 81,000. Scheduling points can be inserted using the same techniques as those that are described above. A point to note here is that the bucket-sort kernel is carefully implemented to make full use of the shared memory per SM. This implementation can lead to severely degraded performance if GLoop runtime uses a small amount of shared memory. We therefore disabled the use of shared memory in the GLoop runtime.

Grep: Grep from the GPUfs project[1] was ported to GLoop. GLoop grep demonstrates that GLoop can support POSIX-like file system APIs since grep reads and writes files stored in the file system of the host operating system. GLoop grep invokes postTask() in each word-search iteration. Every time a string match succeeds, a callback with the file write request is queued to output the result to a file.

Approximate Image Matching: This application (img) from GPUfs [74] scans three databases, each of which has 390MB (25,000 images in total) for learning, and finds images similar to the ones given as queries (2,000 images: 32 MB in total). The original img uses gmmap() and gunmap() APIs in GPUfs, which enable file caches to be placed in the GPU memory. This feature results in significant performance benefits because the same files are accessed repeatedly in img. Our current prototype of GLoop lacks this feature, which results in poorer performance than that of the original. However, this feature is orthogonal to our design of GLoop and can be incorporated into GLoop.

Echo Server: To demonstrate that GLoop can support socket-like APIs for networking, the echo server from the GPUnet project [48] was ported. GLoop provides TCP/IP networking APIs such as accept(), recv(), and send() although the GPUnet assumes RDMA for communication GLoop prepares bounce buffers in the GPU memory to enable DMA between a host and device loops. Every time the echo server invokes networking APIs, which provide scheduling opportunities.

Matrix Multiplication Server: This application (matmul server) from the GPUnet is a network server that multiplies two 256×256 matrices of floats in a tiling manner. Matmul server mimics a typical GPGPU server behavior in which

[1] `https://github.com/gpufs/gpufs`

the GPU consumes the transferred input and sends back the result, such as a face verification server [48]. The invocation of networking APIs provides scheduling opportunities, as is done in the echo server. In addition, every time a tile is calculated, postTaskIfNecessary() is invoked to incorporate more scheduling points.

5.6 Experiments

We conducted experiments to find answers to four questions: (1) how much overhead does GLoop incur, (2) how well does the GLoop application perform when multiple GPU applications are consolidated, (3) can we achieve performance isolation on GLoop, and (4) is GLoop effective in consolidation scenarios?

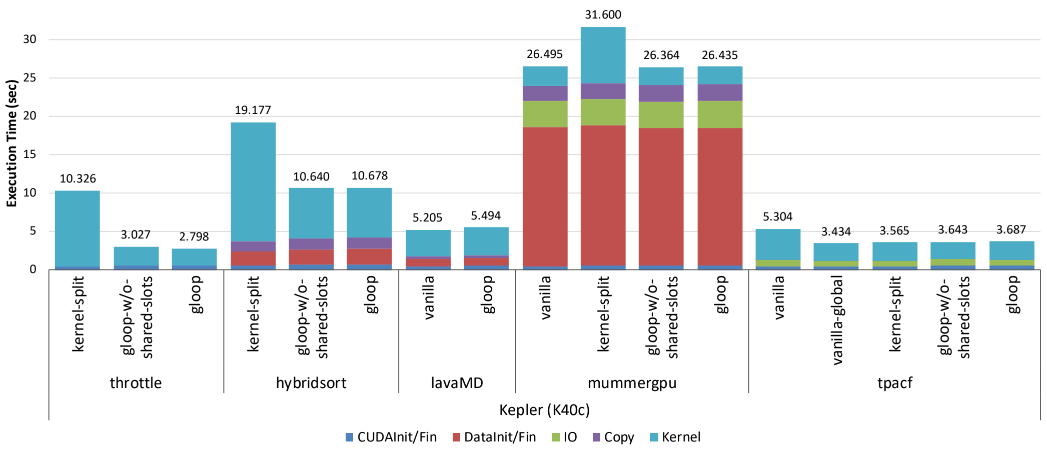
We evaluated our prototype on an MAX-XW-E5HG machine with two Xeon E5-2620 v4 2.10-GHz CPUs (each has eight cores), 64-GB of memory and one 726-GB SSD. We used two NVIDIA Tesla GPUs: K40c Kepler GPU with 12-GB GDDR5 memory and P100 Pascal GPU with 16GB HBM2 memory. The NVIDIA GPU driver version is 384.81. The disk performance reported by hdparm is 8855.98 and 337.46 MB/s for cached and disk reads, respectively.

The workload was executed eleven times, i.e., once to warm up and ten times to obtain results. The measurements reported below are average values of the ten executions.

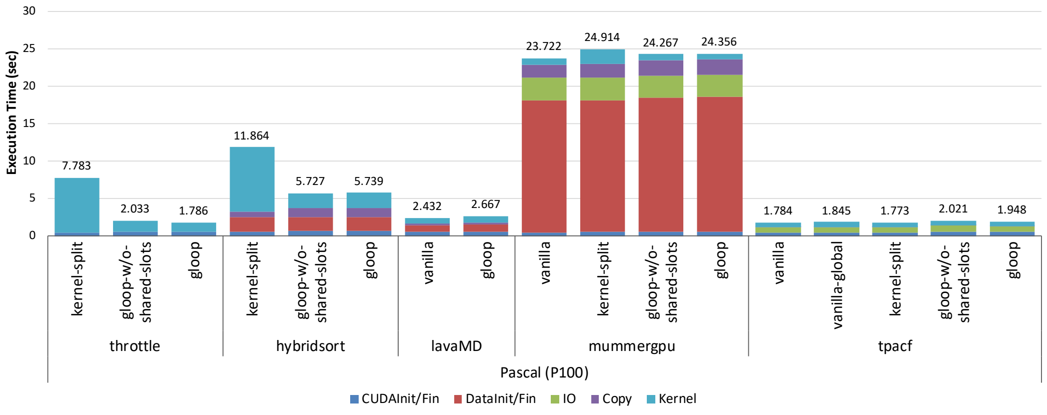
5.6.1 Standalone Overhead

To find how much overhead GLoop incurred, we ran the applications described in Section 5.5 in a standalone manner and measured their execution times. We grouped our applications into two categories: GPU- and I/O-intensive. We discuss GLoop's overhead based on these two groups.

We ran three versions of GPU applications: an unmodified one (vanilla), a GLoop-based one (gloop), and a split version where the original GPU kernels were split into multiple short kernels (kernel-split). We also ran GLoop versions without the shared memory optimization, which were postfixed as -w/o-shared-slots.



(a) On Kepler K40c GPU.



(b) On Pascal P100 GPU.

Figure 5.5: Execution times and their breakdown.

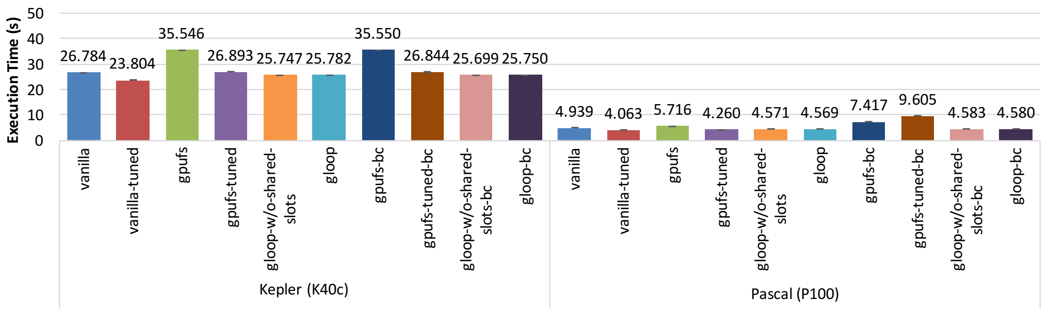


Figure 5.6: Execution times for grep variations.

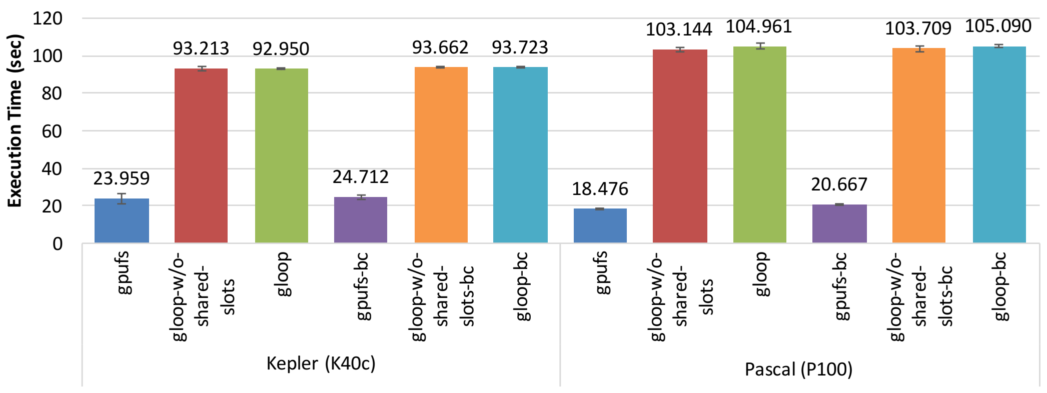


Figure 5.7: Execution times for img variations.

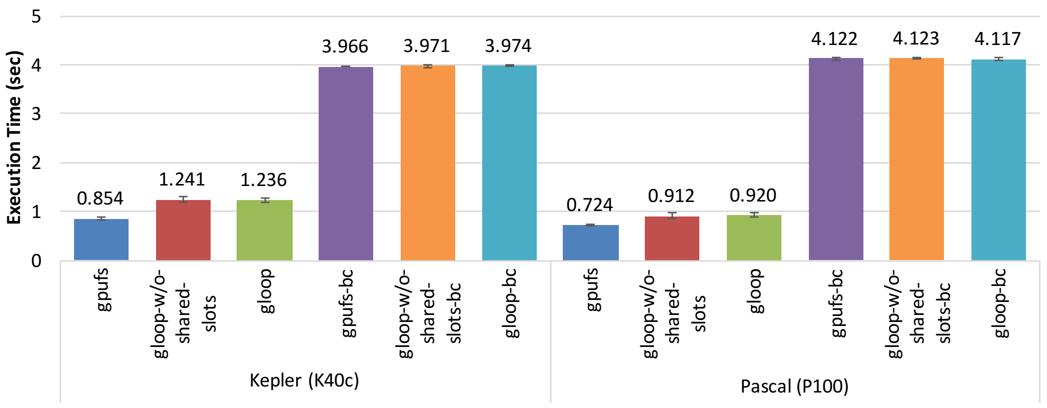


Figure 5.8: Execution times for img-simple variations.

Scheduling Point Latency: To examine the scheduling point latency, we first measured the execution time for the microbenchmark called throttle. Throttle is a program that invokes postTask() one million times with one thread block that consists of one thread. The left end of Figure 5.5a shows the execution time. GLoop version is 3.69× faster than kernel-split in Kepler, because GLoop offers lightweight scheduling points that do not involve kernel launches. GLoop version with the shared memory optimization is 8.2% faster than gloop-w/o-shared-slots, since throttle frequently writes a callback in slots on the shared memory.

On Pascal GPUs, GLoop is more effective than on kernel-split. The left end of Figure 5.5b shows that GLoop version is 4.36× faster than kernel-split in Pascal. This is because time for GPU kernel launches is not changed while the GPU clock of P100 is faster than K40c. This result means that the GPU kernel launch are relatively costly on the Pascal GPU.

From the execution time and number of scheduling points, we estimated that the latency of a scheduling point, which does not include the scheduling algorithm or GPU kernel switch costs, is less than 2.26 μs in Kepler and 1.23 μs in Pascal, calculated by $\frac{ExecutionTimeofGPUKernel}{1000000}$. We can hide this latency since GPU executes different warps when accessing GPU memory. In fact, the subsequent experiments revealed that the scheduling point latency of GLoop is hidden or amortized in application benchmarks.

Compute-intensive Applications: We measured the execution times for hybridsort, lavaMD, mummergpu, and tpacf. We did not split the GPU kernel of lavaMD or the printKernel of MUMmergpu because their thread blocks are short-lived. For laveMD, we did not prepare a -w/o-shared-slots version because it does not use postTask(). For tpacf, we also prepared a vanilla-global version that uses the global memory based on the original one, as described in Section 5.5. While vanilla hybridsort works with CUDA 7.5 [80], it crashes with CUDA 9.0. Thus, we omitted the result from the figure. We divided the total execution time into five categories: CUDAInit/Fin, DataInit/Fin, IO, Copy, and Kernel. They correspond to the times for CUDA context initialization and finalization, constructing and destroying data, reading and writing files, transferring data between the host and device, and GPU kernel execution.

Figure 5.5 presents the results. GLoop's overhead is shown in the CUDAInit/Fin and Kernel categories, and is small (-0.2% – 7.3%). The overhead in the CUDAInit/Fin category comes from allocating additional GPU memory and threads for GLoop. The time is small (539 ms – 725 ms) compared to the Kernel category (2258 ms – 6506 ms). Since this is one time overhead during GPU application execution, it is amortized by executing GPU kernels for long time; such GPU kernels are our target application. The overhead in the Kernel category is small or negligible (-10.5% – 4.3%) in all the cases except for hybridsort. The other categories are similar in all cases. The kernel performance penalty of GLoop is 23.8% in hybridsort, compared to CUDA 7.5 hybridsort data reported in our previous work [80]. This is caused by the balance of the two GPU kernel executions. The bucket-sort kernel is faster in the non-shared mode because of its shared memory utilization. The merge-sort kernel' performance, on the other hand, is better in the shared mode because it does not use shared memory. All the kernels in one application are currently compiled in either the shared or the non-shared mode due to limitations in the toolchain. We can extend GLoop to mitigate the overhead by changing this mode per GPU kernel.

GLoop outperforms kernel-split versions in almost all cases. It is 1.8× and 1.2× faster than the kernel-split of hybridsort and mummergpu, respectively. The kernel-split versions cause numerous kernel launches for scheduling, whereas GLoop does not involve such additional kernel launches. In addition, GLoop reduces callback saves and restores by using postTaskIfNecessary(). The execution time of the kernel-split version in tpacf is comparable to that of GLoop because the execution time of each kernel is sufficiently long to amortize the kernel launch overhead.

The kernel execution time on Pascal is faster than that on Kepler. The main reason of this speed up is for the number of SMs: Pascal P100 has 3.73× more SMs (56) than Kepler K40c (15). On Pascal, the execution time of GLoop is comparable to that of vanilla and is much faster than that of kernel-split in several benchmarks (2.07× in hybridsort and 1.02× in mummergpu).

We can also see an interesting performance trend that time for GPU kernel launches is almost the same among two GPUs, which means that the overhead of GPU kernel launches is bigger on P100 than Kepler. If the GPU kernel launch incurs the same overhead while GPUs become faster, its relatively worse overhead motivates developers to consolidate numerous computation into one GPU kernel to minimize the GPU kernel launches. The optimization of merging tiny GPU kernels that is employed in a recent research system [36] can be used to reduce GPU kernel launches.

I/O-intensive Applications: We measured the execution times of grep and img. We prepared GPUfs- and GLoop-based versions labeled gpufs and gloop, and prepared a workload for comparison that pre-allocates a large amount of GPU device memory to transfer all the datasets before starting the GPU kernel, called vanilla. The execution time measured just after the host buffer cache is cleared is postfixed as -bc. We tuned gpufs and vanilla in grep to gain further CUDA occupancy with our GPU. These tuned versions are postfixed as -tuned. We modified the source code of the downloaded GPUfs to run it on our GPU.

The results obtained for grep are in Figure 5.6. The performance of gloop is comparable to the other grep implementations on K40c and P100. Gloop in K40c outperforms the untuned versions and is 7.7% slower than the vanilla-tuned and 4.3% faster than gpufs-tuned versions.

Clearing the buffer cache does not degrade GLoop performance (-0.1% in K40c and 0.2% in P100). Since grep repeatedly reads the same set of files, no buffer cache misses occur except for the first read. The execution time of gpufs-bc and gpufs-tuned-bc on P100 is slower than that of gloop. This comes from the difference of I/O handling between them. Since P100 GPU core performance is better than K40c, I/O throughput becomes a relatively major factor in the grep execution. We believe that the GLoop's multi-threaded asynchronous I/O feature maximizes I/O throughput in grep. The GPUfs implementation executes only one I/O thread in the host-side and thus fails to fully utilize the disk bandwidth.

Figure 5.7 shows the results for img. Since img-gpufs and img-gpufs-bc occasionally crash with CUDA 9.0 SDK, we measure the average execution time of first eleven successful execution. The execution time for gloop is 3.88× and 5.68× longer than that for gpufs in K40c and P100. This is because gpufs can benefit from the GPUfs' GPU buffer cache: GPUfs builds its buffer cache in the GPU device memory, and thus, the cache works effectively as the workload repeatedly reads the same files.

We used another data set called img-simple to validate this expectation. Img-simple uses only one image as the query data, runs one thread block, and never provides matches against dataset images. This avoids reading the same data from the file system multiple times and reduces the effect of the GPU buffer cache as much as possible.

The results in Figure 5.8 indicate that the execution time of gloop is just 1.45× and 1.27× longer than that of gpufs in K40c and P100 respectively. The remaining overhead is caused by the additional data copies in the gloop version. While gpufs exposes mmap-like APIs that only cause one data copy from the host to the GPU buffer cache, gloop provides write/read like APIs that perform copies twice, from the host memory to the GPU bounce buffer, and from the bounce buffer to the GPU user buffer. This overhead can be eliminated by adding a gpufs-like buffer cache mechanism to the GLoop runtime. This implementation just requires engineering

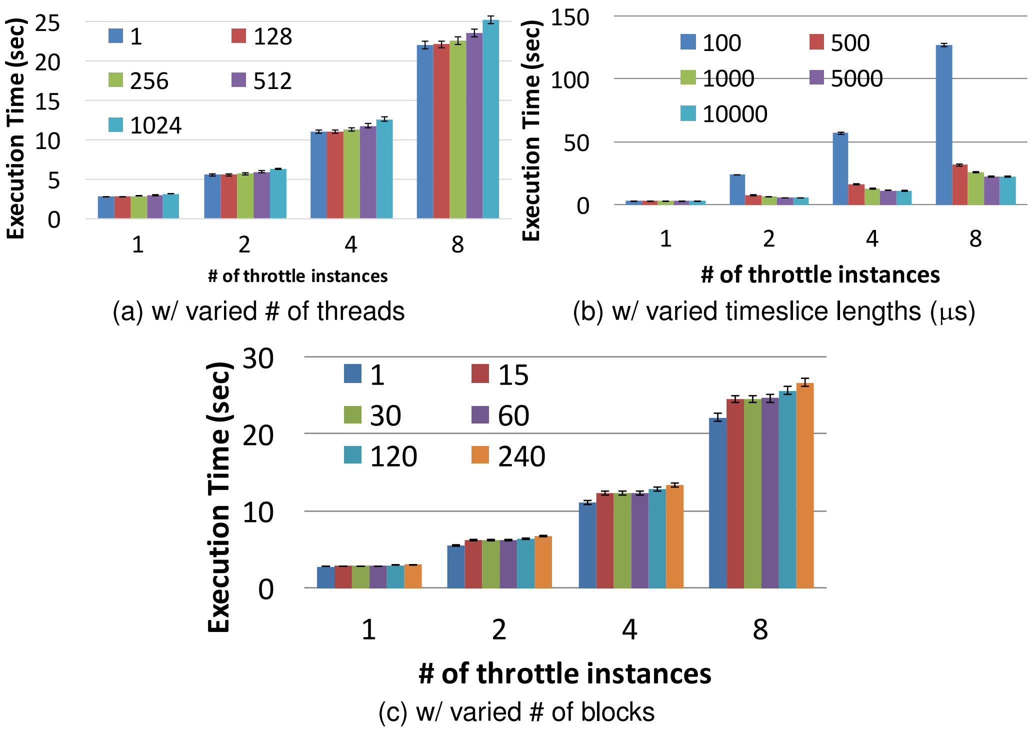


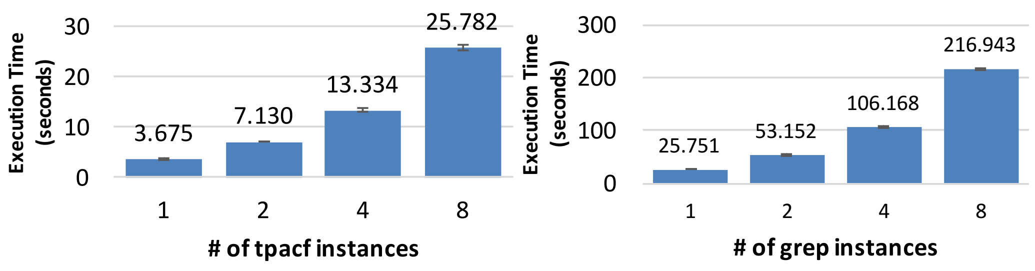
Figure 5.9: Performance across multiple throttle instances.

effort but a description of the implementation of a buffer cache is beyond the scope of this dissertation.

Different from the grep case, the cold buffer cache degrades performance by 0.8% in K40c and 0.1% in P100 because img issues I/O requests more frequently than grep.

5.6.2 Performance at Scale

We concurrently ran multiple instances of GLoop-based applications to find the performance penalty of GLoop's consolidation. We launched one, two, four, and eight instances and measured each of their execution times. We used throttle as the microbenchmark, tpacf as a compute-intensive application, and grep as an I/O-intensive application. We measured the execution time of these benchmarks on K40c GPU.



(a) tpacf (b) grep

Figure 5.10: Performance across multiple application instances.

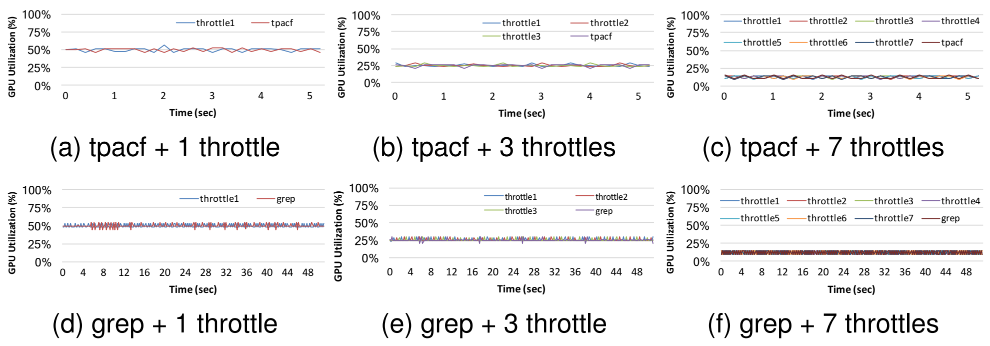


Figure 5.11: GPU utilization for GPU applications (over 200 ms).

Context Switching Latency: To discern the context switching overhead, we ran throttle by varying the number of threads, timeslice lengths, and the number of thread blocks. Figure 5.9a plots the execution times for different numbers of threads. In the single instance case, the increase in threads from 1 to 1024 lengthens the execution time by 13.6%. This is because the scheduling points require thread synchronization. The tendency in the case of eight instances is almost the same as that in the single instance (14.3% longer execution time from 1 to 1024). The slight overhead results from thread synchronization done in context switching.

Figure 5.9b plots the results for varied timeslice lengths. Due to the optimization of GLoop to avoid polling on the PCIe bus, as described in Section 5.2.2, the actual timeslice consumed slightly differs from the specified value. GLoop performs 2636 context switches on average per application in the two instances with

10000 µs timeslices in eleven runs. Thus, throttle in the two instances performs 239.6 context switches for each execution on average. When the timeslices are too short, context switches are dominant in the execution times. The execution time with a 100 µs timeslice is 5.8× longer than that with a 10000 µs timeslice for eight instances, where the execution time with the 100 µs timeslice is 42.5× longer than that of the one instance. The longer timeslice mitigates the context switching overhead. The increase in execution time for the 10000 µs timeslice is linear from one to eight instances (7.88×).

Figure 5.9c shows the execution time for throttle with different number of thread blocks. The execution time does not increase linearly up to 240 since K40c GPU can execute 240 thread blocks of throttle concurrently. However, in the eight instance case, the execution time with 240 thread blocks is 20.3% longer than that with 1 thread block. This time increase comes from the limitation of GLoop's suspend mechanism. The thread block scheduler stops the running thread blocks asynchronously; it checks the suspend signal only every time slice and thus some GPU cores used by suspended thread blocks become idle until all the thread blocks are suspended. Suspending more running thread blocks causes more idle cores in the GPU kernel suspension, leading to waste computing resource of a GPU.

Applications: Figure 5.10 presents the results obtained for tpacf and grep. The x-axis in the figures represents the number of launched applications, and the y-axis represents the execution time. GLoop schedules the applications in a fair-share manner, and the standard deviation for the results is at most 2.5%.

The figure indicates that the execution time for tpacf applications increases in proportion to the number of instances. The execution time for eight instances is slightly better than eight times the standalone's execution time because of the short I/O time in tpacf. From Figure 5.5, tpacf performs file I/O, which can be overlapped with execution of the other tpacf kernels.

The execution times for eight grep instances exceed eight times the standalone's execution time (8.43×) as a result of disk I/O contention. The result in the next section validates this finding: the execution time for grep with seven throttles is 8.28× longer than that for the standalone since throttle does not issue any I/O requests. The remaining slowdown stems from the overhead imposed by scheduling the running instances.

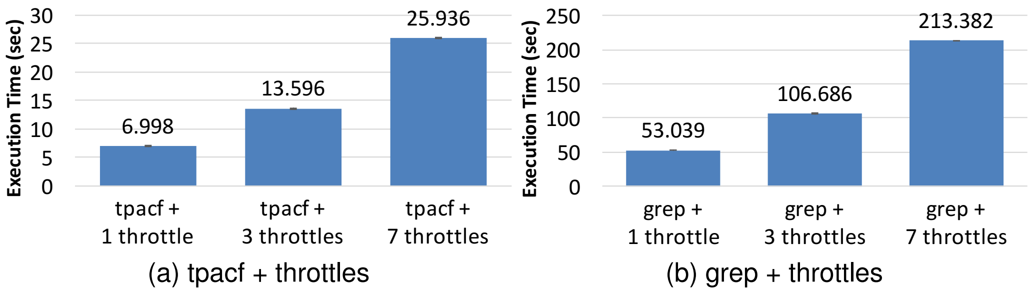


Figure 5.12: Execution times for GPU applications with throttles.

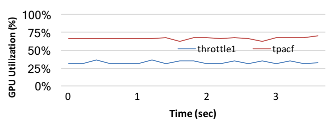
5.6.3 Performance Isolation

We demonstrated that GLoop isolates performance among GPU applications. We controlled the GPU utilization of consolidated applications by using our proportional scheduler and measured the GPU utilization of each application. We launched one GPU application instance. We used tpacf and grep as GPGPU applications. We also ran one, three, and seven instances of throttle together.

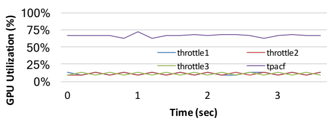
First, we ran all the applications with the same utilization assignment. Figure 5.11 plots the GPU utilization. The x-axis represents the elapsed time, and the y-axis is the GPU utilization of the applications over 200 ms. The figure reveals that GLoop achieves performance isolation in all cases. The applications share one GPU and the computation resources are fairly divided. When running two, four, and eight instances, their GPU utilizations correspond to 50%, 25%, and 12.5%.

Figure 5.12 shows the execution times for each instance. The execution time for eight instances for tpacf is 3.71× longer than that for the two instances. The increase in the execution time is not linear since the short I/O time in tpacf is constant in all cases. The execution time for grep in the eight instances is 4.02× longer than in two instances. This results from the overhead for scheduling multiple GPU applications.

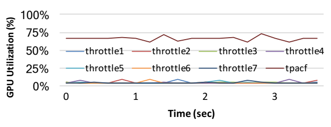
Next we changed the resources assigned to the GPU applications. We assigned 66% of the utilization to a target application (tpacf or grep) while co-running throttle instances shared the GPU with one another. Figure 5.13 plots the results, which reveal that GLoop successfully assigns a target application the weighted GPU utilization and the other throttles share the remaining resources. The execution time



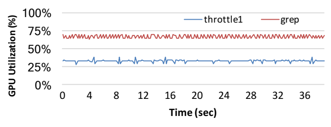
(a) tpacf + 1 throttle



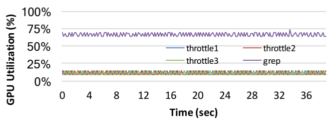
(b) tpacf + 3 throttles



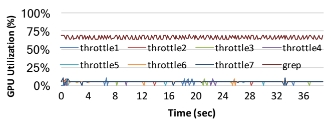
(c) tpacf + 7 throttles



(d) grep + 1 throttle

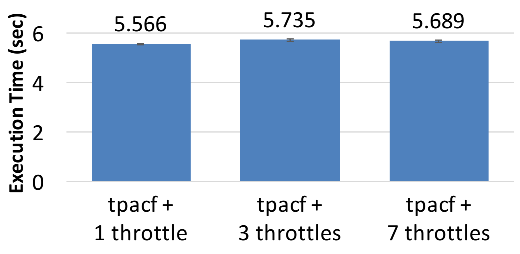


(e) grep + 3 throttle

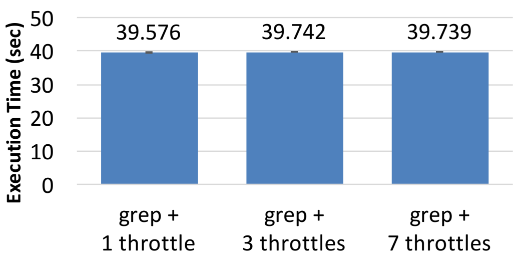


(f) grep + 7 throttles

Figure 5.13: GPU utilization for GPU applications (over 200 ms). 66% of the GPU resources is assigned to the applications.



(a) tpacf + throttles



(b) grep + throttles

Figure 5.14: Execution times for GPU applications with throttles. 66% of the GPU resources is assigned to the target application.

shown in Figure 5.14 slightly increases with the number of applications due to the accumulated overhead of the scheduler.

5.6.4 Consolidation Scenarios

To confirm the effectiveness of consolidating GLoop's GPU applications, we devised two scenarios: GPU Server Consolidation and GPU Idle-time Exploitation. The GPU server consolidation is a situation where under-utilized GPU servers are consolidated into one GPU, while the GPU idle-time exploitation is where a compute-intensive application exploits the idle time of an under-utilized GPU.

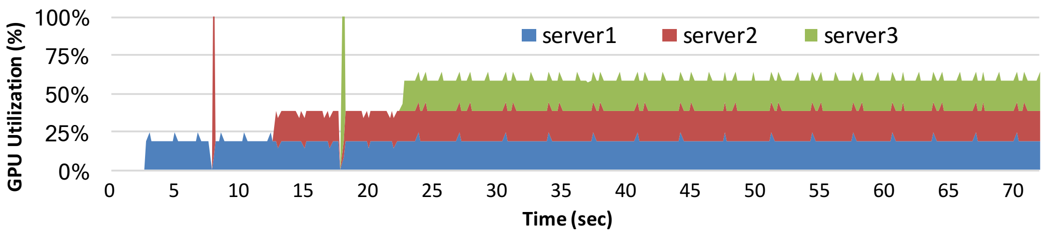


Figure 5.15: Stacked GPU utilization for consolidated GPU matmul servers (over 200 ms).

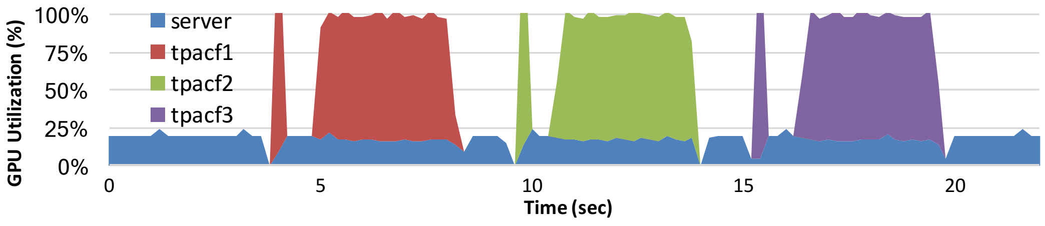


Figure 5.16: Stacked GPU utilization for GPU matmul server and tpacf application (over 200 ms).

GPU Server Consolidation: To demonstrate that GLoop successfully consolidates under-utilized GPU servers on a single GPU, we configured the applications as follows. We first ran an under-utilized GPU matmul server (server1) on a GPU whose utilization was roughly 20%. Then we gradually launched two additional under-utilized GPU servers (server2 and server3) on the same GPU. When a single server was running, GLoop assigned it 100% of GPU utilization for the polling of device loops. To clearly demonstrate the effectiveness of GLoop's consolidation, we launched a low priority throttle to drain the remaining utilization.

Figure 5.15 shows the stacked GPU utilization per 200 ms. The x-axis indicates the stacked GPU utilization of the three under-utilized GPU servers, and the y-axis plots the time series. While the new servers are being launched, GLoop successfully maintains the GPU utilization of the running servers at 20%. The figure also plots that the resource utilization spikes when server2 and server3 start (at the points of 7.5 and 17.5 s). This is because the CUDA initialization (CUDAInit/Fin in Section 5.6.1) takes 500 ms and thus GPU utilization is temporarily occupied. We note that it was 200 ms in 361.42 GPU driver. We guess that the internal initialization procedure is changed.

GPU Idle-time Exploitation: We also demonstrated that GLoop effectively assigns idle resources to the compute-intensive application while maintaining the performance of low utilized GPU servers. We ran one under-utilized GPU server (server) and then sequentially launched tpacf (tpacf1, tpacf2, and tpacf3). As is the same to the previous experiment, we launch a throttle application draining the remaining utilization too.

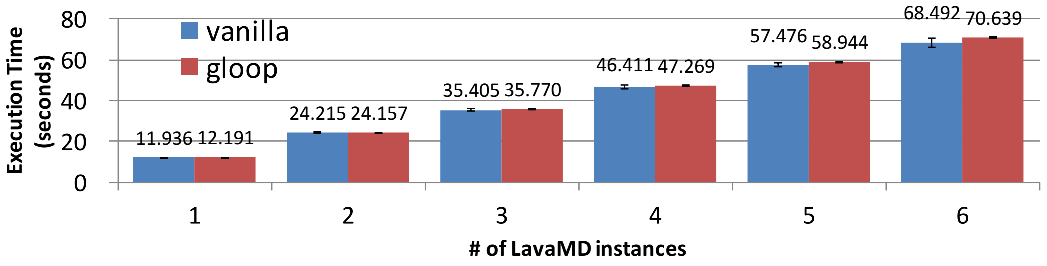
The results are plotted in Figure 5.16. When we launch a tpacf instance, the tpacf first occupies the GPU for its initialization. After that, while the server process runs under the assigned resources, tpacf utilizes the rest of the GPU resources. GLoop successfully assigns idle GPU utilization to tpacf instances while the resource utilization of the server is preserved.

5.6.5 Hardware Preemption

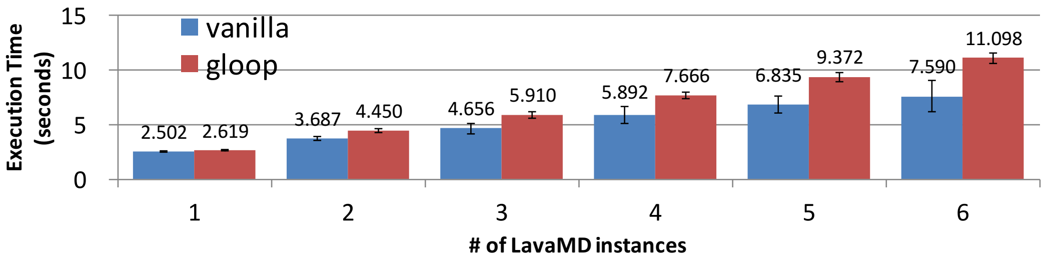
Finally, we describe an anecdotal situation showing that our software-level preemption can be more effective than hardware preemption. Pascal's compute preemption offers instruction-level granularity preemption. The preemption mechanism saves/restores context information on the GPU kernels to/from GPU DRAM. Since the context information includes thousands of registers' values and large shared memory contents, the context switching could cause high latency. On the other hand, GLoop allows developers to insert scheduling points at appropriate places where the size of the context information becomes small. For example, we do not need to save register values and shared memory content as context information at a scheduling point when a thread block finishes. While the strength of GLoop is that it can offer flexible software scheduling control, the above situation implies our approach can achieve efficient context switches.

To validate the above assumption, we run LavaMD benchmark on two Pascal GPUs, GTX 1080 and Tesla P100 Pascal GPUs. In each trial, we launch multiple instances of vanilla and GLoop-based LavaMD (1 to 6) and measure their execution time.

Figure 5.17 shows the average execution time and standard deviation. The standalone performance shows that GLoop causes 2.1% in GTX 1080 and 4.7% in P100 performance penalties stemming from its runtime overhead. We believe this penalty would be further mitigated once we optimize GLoop for Pascal GPUs. On



(a) w/ Pascal GTX 1080 GPU



(b) w/ Pascal P100 GPU

Figure 5.17: Execution times of LavaMD with GLoop and hardware preemption.

the other hand, the cases of two or more instances show that the GLoop version is comparable the vanilla version in GTX 1080 (-0.2 – 3.1%). The standard deviation of the vanilla version is large in the four, five, and six instance cases. This is because the vanilla LavaMD execution is so short that the LavaMD sometimes completes without any hardware preemption. This completion is observed more frequently in P100. GLoop can schedule applications stably with its software mechanism.

The overhead of GLoop is higher on P100 (4.7 – 46.2%) than that of GTX 1080 due to the difference of the LavaMD configurations, each of which is tuned to each GPU. The major difference is the number of the running thread blocks. In LavaMD, P100 executes more thread blocks at once than GTX 1080 since the number of SMs on P100 is 2.8× more than that on GTX 1080. This causes two negative impacts related to the overhead on P100. First, the standalone performance of LavaMD on P100 is worse than that on GTX 1080 and thus the performance penalty on P100 is relatively bigger. Second, LavaMD on P100 launches more thread blocks and causes context switching overhead shown in Section 5.6.2. We could mitigate this overhead by a NVIDIA Volta GPU's feature [62]: statically partitioning SMs to lower the number of the running thread blocks and switching the running thread blocks eventually for each SM.

This result shows that software-level approach can potentially perform efficient context switches compared with hardware preemption in some situations with flexible scheduling control.

5.7 Summary

This chapter presented GLoop, a pure software runtime that allows us to consolidate GPGPU applications including GPU eaters on a shared GPU. GLoop offers an event-driven programming model so that we can develop highly functional GPGPU applications and schedule them on a shared GPU in a fine-grained manner. The GLoop runtime executes the GPU kernels that are isolated from one another, provides lightweight scheduling points, and schedules them according to a proportional share scheduling policy. In addition, it runs on a proprietary GPGPU software stack including the NVIDIA driver and CUDA library. We implemented a prototype of GLoop and ported eight GPU eaters on it. The experimental results demonstrate that our prototype efficiently consolidates GPGPU applications.

Recent adoption of binary-offered GPU kernels such as NVIDIA cuDNN poses an issue of GLoop applicability; their proprietary nature means that we cannot modify them. One of our future work will be to explore ways to transform GPU kernels including binary blobs into GLoop-based applications. One possible direction is to transform GPU applications into GLoop-based applications and insert scheduling points by using GPU binary analysis frameworks [20].

Completely automatic transformation into GLoop-based applications is also challenging. This is because we cannot estimate the execution time of a specific part of the application from the static information. Thus, profiling-based or semi-automatic insertion techniques are promising. One solution is inserting many scheduling points into the program mechanically, taking profiling information, and removing unnecessary scheduling points. Another is that programmers can specify which scheduling points are necessary by referring to the reported profiling information. This mechanism can further reduce the programmer effort.

Chapter 6

Conclusion

6.1 Contribution Summary

This dissertation has conducted studies of GPU resource virtualization in the multi-tenant cloud environments. GPUvm clarifies the tradeoffs between GPU virtualization approaches. GLoop presents an application-assisted approach that can host GPU eaters in the cloud environments.

GPUvm showed the tradeoffs between GPU virtualization approaches in terms of software modifications, the levels of interfaces, and performance. This dissertation presented the design and implementation of full-virtualization of GPUs. The analysis of the experimental results showed that the full-virtualization incurs significant overhead, and the bottleneck of the full-virtualization is largely the cost of the MMIO handling and page table shadowing. This dissertation demonstrated that the high-level interfaces to the virtual GPUs can mitigate these overheads at the expense of modification of the device driver; the naive para-virtualization removing the page table shadowing improved the performance and the high-performance para-virtualization reduced the MMIO handling overhead. This result helps the cloud software developers to select an appropriate virtualization approach for their use cases.

Although application-transparent approaches including GPUvm can host various classes of GPGPU applications, hosting GPU eaters are challenging for these approaches because they schedule GPGPU applications at a coarse-grained boundary, GPU commands or kernels. This dissertation demonstrated that

GLoop, an application-assisted approach, can schedule GPU eaters on a shared GPU according to a proportional share scheduling policy, by introducing event-driven programming model into GPGPU kernels. GLoop allows the multi-tenant clouds to share GPUs with a wider range of applications including GPU eaters.

6.2 Future Directions

GPUvm has uncovered that the largest bottleneck of the full-virtualized GPUs is the page table shadowing. Thus, one of the future work is to reduce or eliminate this overhead. For example, designing a nested page table support in GPU hardware as a virtualization extension, similar to Intel EPT, is effective to reduce overhead of GPU full-virtualization. The nested page table translates GPU guest physical addresses to GPU host physical address, so that the guest GPU driver can use the unmodified guest GPU page tables, which eliminates the need of shadowing. While the nested page table introduces an additional cost of the two-level page walk, effective use of huge pages in GPU memory could mitigate this overhead.

Automatic transformation into GLoop-based applications is another one of the future work. While we insert lightweight scheduling points of GLoop manually, profiling the runtime data could advise developers where to insert lightweight scheduling points. This semi-automatic approach can reduce the burden of making GLoop-based applications and expand applicability of GLoop.

Introducing an abstraction layer for GPUs like GPUvm and GLoop can simplify the management and improve the availability of GPUs in the multi-tenant cloud environments. Virtualized GPU contexts pave the way to allowing hypervisors to migrate VMs using GPUs by migrating virtual GPU contexts among physical GPUs. Another use case is a high availability system that keeps the state of the secondary virtual GPU in synchronization with the primary one and makes the secondary one active when the primary one becomes unavailable.

While this dissertation focused on discrete GPUs, the design of GLoop could be applicable to the other types of simple accelerators that do not support preemptions (low-end GPUs, FPGAs etc.). The GLoop design envisions the future cloud environments making wider range of accelerators sharable.

Appendix A

GLoop APIs

Table A.1: List of GLoop APIs in our prototype.

API	**Explanation**
fs::open / fs::close	Open and close a file descriptor.
fs::fstat	Retrieve size information of a given file.
fs::ftruncate	Truncate a file to a given size.
fs::read / fs::write	Read from and write to a file at a given offset.
net::connect	Open a TCP socket for client.
net::bind / net::unbind	Open and close a server.
net::accept	Open a TCP socket for a incoming connection on a given server.
net::close	Close a TCP socket.
net::receive / net::send	Receive from and send to a socket.
loop::postTask	Post a given callback to the event loop.
loop::postTaskIfNecessary	Post a given callback to the event loop if the event loop needs to check the suspend signal.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation*, pages 265–283. USENIX, 2016.

[2] D. Abramson. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(3):179–192, 2006.

[3] S. R. Agrawal and A. R. Lebeck. Rhythm : Harnessing Data Parallel Hardware for Server Workloads. In *Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 19–34. ACM, 2014.

[4] T. Aila and S. Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. of the Conf. on High Performance Graphics*, pages 145–149. ACM, 2009.

[5] A. M. Aji, A. J. Peña, P. Balaji, and W.-c. Feng. MultiCL: Enabling automatic scheduling for task-parallel workloads in OpenCL. *Parallel Computing*, 58:37–55, 2016.

[6] Amazon.com. EC2: Amazon Elastic Compute Cloud.

[7] AMD. AMD Kaveri. `http://www.amd.com/en-us/products/processors/desktop/a-series-apu`.

[8] AMD. AMDGPU. `http://cgit.freedesktop.org/~agd5f/linux/log/?h=amdgpu`.

[9] A. Amiri Sani, K. Boos, S. Qin, and L. Zhong. I/O Paravirtualization at the Device File Boundary. In *Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 319–332. ACM, 2014.

[10] N. Amit and M. Ben-Yehuda. vIOMMU: efficient IOMMU emulation. In *Proc. of the 2011 USENIX Annual Technical Conf.*, pages 73–86. USENIX, 2011.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and Art of Virtualization. In *Proc. of the 19th Symp. on Operating Systems Principles*, pages 164–177. ACM, 2003.

[12] C. Basaran and K.-D. Kang. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *Proc. of the 24th Euromicro Conf. on Real-Time Systems*, pages 287–296. IEEE, 2012.

[13] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*. USENIX, 2014.

[14] S. Breß, H. Funke, and J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *Proc. of the 2016 Int'l Conf. on Management of Data - SIGMOD '16*, pages 1891–1906, New York, New York, USA, 2016. ACM.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. of the 2009 Int'l Symp. on Workload Characterization*, pages 44–54. IEEE, 2009.

[16] G. Chen, Y. Zhao, X. Shen, and H. Zhou. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In *Proc. of the 22nd*

ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, pages 3–16. ACM, 2017.

[17] L. O.-S. Community. Memory Mapped I/O Trace. `https://nouveau.freedesktop.org/wiki/MmioTrace/`.

[18] L. O.-S. Community. Nouveau Open-Source GPU Device Driver. `http://nouveau.freedesktop.org/`.

[19] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proc. of the 11th European Conf. on Computer Systems*, pages 1–16. ACM Press, 2016.

[20] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *Proc. of the 19th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 353–364. ACM, 2010.

[21] Y. Dong, Z. Yu, and G. Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *Proc. of the 1st Workshop on I/O Virtualization*, pages 10–16. USENIX, 2008.

[22] M. Dowty and J. Sugerman. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[23] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. of the 2010 Int'l Conf. on High Performance Computing & Simulation*, pages 224–231. IEEE, 2010.

[24] N. Foundation. Node.js, 2016. `https://nodejs.org`.

[25] K. Fraser, S. H, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, pages 1–10. ACM, 2004.

[26] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A GPGPU transparent virtualization component for high performance computing clouds. In *Proc. of the 16th Int'l Euro-Par Conf. on Parallel Processing*, pages 379–391. Springer-Verlag, 2010.

[27] A. Goswami, J. Young, K. Schwan, N. Farooqui, A. Gavrilovska, M. Wolf, and G. Eisenhauer. GPUShare: Fair-Sharing Middleware for GPU Clouds. In *2016 IEEE Int'l Parallel and Distributed Processing Symp. Workshops*, pages 1769–1776. IEEE, 2016.

[28] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. LoGV: Low-overhead GPGPU Virtualization. In *Proc. of the 4th Int'l Workshop on Frontiers of Heterogeneous Computing*, pages 1721–1726. IEEE, 2013.

[29] A. G. Greenberg and N. Madras. How fair is fair queuing. *Journal of the ACM*, 39(3):568–598, 1992.

[30] K. Gupta, J. A. Stuart, and J. D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Proc. of the Innovative Parallel Computing*, pages 1–14. IEEE, 2012.

[31] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-Accelerated Virtual Machines. In *Proc. of the 3rd Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.

[32] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *Proc. of the 2011 USENIX Annual Technical Conf.*, pages 31–44. USENIX, 2011.

[33] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-Accelerated Software Router. In *Proc. of the ACM SIGCOMM 2010 Conf.*, pages 195–206. ACM, 2010.

[34] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational Joins on Graphics Processors. In *Proc. of the 2008 ACM SIGMOD Int'l Conf. on Management of Data*, pages 511–524. ACM, 2008.

[35] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proc. of the 6th ACM Symp. on Cloud Computing*, pages 43–57. ACM, 2015.

[36] M. Hidaka, Y. Kikura, Y. Ushiku, and T. Harada. WebDNN: Fastest DNN Execution Framework on Web Browser. In *Proc. of the 2017 ACM on Multimedia Conf.*, pages 1213–1216. ACM, 2017.

[37] M. Hirabayashi, S. Kato, M. Edahiro, K. Takeda, T. Kawano, and S. Mita. GPU Implementations of Object Detection using HOG Features and Deformable Models. In *Proc. of the 1st Int'l Conf. on Cyber-Physical Systems, Networks, and Applications*, pages 106–111. IEEE, 2013.

[38] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*, pages 1–14. USENIX, 2011.

[39] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proc. of the 22nd ACM Int'l Conf. on Multimedia*, pages 675–678, New York, New York, USA, 2014. ACM.

[40] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU Join Processing Revisited. In *Proc. of the 8th Int'l Workshop on Data Management on New Hardware*, pages 55–62. ACM, 2012.

[41] S. Kato, J. Aumiller, and S. Brandt. Zero-copy I/O processing for low-latency GPU computing. In *Proc. of the 4th Int'l Conf. on Cyber-Physical Systems*, pages 170–178. ACM/IEEE, 2013.

[42] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *Proc. of the 32nd Real-Time Systems Symp.*, pages 57–66. IEEE, 2011.

[43] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proc. of the 2011 USENIX Annual Technical Conf.*, pages 17–30. USENIX, 2011.

[44] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc. of the 2012 USENIX Annual Technical Conf.*, pages 401–412. USENIX, 2012.

[45] J. Kehne, J. Metter, and F. Bellosa. GPUswap: Enabling Oversubscription of GPU Memory through Transparent Swapping. In *Proc. of the 11th ACM Int'l Conf. on Virtual Execution Environments*, pages 65–77. ACM, 2015.

[46] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proc. of the 2010 Int'l Conf. on Management of Data*, pages 339–350. ACM, 2010.

[47] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proc. of the 26th ACM Int'l Conf. on Supercomputing*, pages 341–352. ACM, 2012.

[48] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*, pages 201–216. USENIX, 2014.

[49] M. Koscielnicki. Envytools. `https://0x04.net/envytools.git`.

[50] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-Independent Graphics Acceleration. In *Proc. of the 3rd ACM Int'l Conf. on Virtual Execution Environments*, pages 33–43. ACM, 2007.

[51] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. HippogriffDB: balancing I/O and GPU bandwidth in big data analytics. In *Proc. of the VLDB Endowment*, volume 9, pages 1647–1658. VLDB Endowment, oct 2016.

[52] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 11:1 – 11:12. ACM, 2011.

[53] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee. Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice. In *Proc. of the 2011 Int'l Conf. on Robotics and Automation*, pages 4889–4895. IEEE, 2011.

[54] K. Menychtas, K. Shen, and M. L. Scott. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack. In *Proc. of the 2013 USENIX Annual Technical Conf.*, pages 291–296. USENIX, 2013.

[55] K. Menychtas, K. Shen, and M. L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 301–316. ACM, 2014.

[56] NVIDIA. NVIDIA's next generation CUDA computer architecture: Fermi, 2009. `http://www.nvidia.com/`.

[57] NVIDIA. NVIDIA's next generation CUDA computer architecture: Kepler GK110, 2012. `http://www.nvidia.com/`.

[58] NVIDIA. CUDA Pro Tip: Occupancy API Simplifies Launch Configuration, 2014. `https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/`.

[59] NVIDIA. GPU-Based Deep Learning Inference: A Performance and Power Analysis, 2015. `http://developer.download.nvidia.com/embedded/jetson/TX1/docs/jetson_tx1_whitepaper.pdf`.

[60] NVIDIA. Multi-Process Service, 2015. `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[61] NVIDIA. NVIDIA Tesla P100 – The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU, 2016. `http://www.nvidia.com/object/pascal-architecture-whitepaper.html`.

[62] NVIDIA. NVIDIA TESLA V100 GPU ARCHITECTURE, 2017.

[63] NVIDIA. Parallel Thread Execution ISA Version 6.2, 2018. `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html`.

[64] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, page 407. ACM, 2013.

[65] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proc. of the 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 593–606. ACM, 2015.

[66] N. Rath, J. Bialek, P. J. Byrne, B. DeBono, J. P. Levesque, B. Li, M. E. Mauel, D. A. Maurer, G. A. Navratil, and D. Shiraki. High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak. *Fusion Engineering and Design*, pages 1895–1899, 2012.

[67] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proc. of the 23rd Symp. on Operating Systems Principles*, pages 233–248. ACM, 2011.

[68] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a Compiler and Runtime for Heterogeneous Systems. In *Proc. of the 24th Symp. on Operating Systems Principles*, pages 49–68. ACM, 2013.

[69] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proc. of the 2010 Int'l Conf. on Management of Data*, pages 351–362. ACM, 2010.

[70] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney. High-throughput sequence alignment using Graphics Processing Units. *BMC bioinformatics*, 8(1):474, 2007.

[71] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation*, pages 671–688. USENIX, 2016.

[72] L. Shi, H. Chen, J. Sun, and K. Li. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, 61(6):804–816, 2012.

[73] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka. Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer. In *Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 3:1–3:11. ACM, 2011.

[74] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a File System with GPUs. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 485–498. ACM, 2013.

[75] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.

[76] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Technical Report IMPACT-12-01*, 2012.

[77] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU Clusters. In *Proc. of the 2011 Int'l Parallel & Distributed Processing Symp.*, pages 1068–1079. IEEE, 2011.

[78] W. Sun, R. Ricci, and M. L. Curry. GPUstore. In *Proc. of the 5th Annual Int'l Systems and Storage Conf.*, pages 1–12. ACM, 2012.

[79] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *Proc. of the 2014 USENIX Annual Technical Conf.*, pages 109–120. USENIX, 2014.

[80] Y. Suzuki, H. Yamada, S. Kato, and K. Kono. Gloop: An event-driven runtime for consolidating gpgpu applications. In *Proc. of the 2017 Symp. on Cloud Computing*, pages 80–93. ACM, 2017.

[81] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *Proc. of the 41st Int'l Symp. on Computer Architecture*, pages 193–204. ACM/IEEE, 2014.

[82] K. Tian, Y. Dong, and D. Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In *Proc. of the 2014 USENIX Annual Technical Conf.*, pages 121–132. USENIX, 2014.

[83] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting Concurrent Kernel Execution on Graphic Processing Units. In *Proc. of the Int'l Conf. on High Performance Computing and Simulation*, pages 24–32. IEEE, 2011.

[84] B. Wu, X. Liu, X. Zhou, and C. Jiang. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proc. of the 22nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 483–496. ACM, 2017.

[85] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W.-c. Feng. VOCL: An optimized environment for transparent virtualization of graphics processing units. In *Innovative Parallel Computing*, pages 1–12. IEEE, 2012.

[86] M. Yu, C. Zhang, Z. Qi, J. Yao, Y. Wang, and H. Guan. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *Proc. of the 22nd Int'l Symp. on High-performance Parallel and Distributed Computing*, pages 203–214. ACM, 2013.

[87] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of processing data warehousing queries on GPU devices. In *Proc. of the VLDB Endowment*, volume 6, pages 817–828. VLDB Endowment, aug 2013.

[88] L. Zeno, A. Mendelson, and M. Silberstein. GPUpIO: The Case for I/O-Driven Preemption on GPUs. In *Proc. of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 63–71. ACM, 2016.